

ECEN-5623
ASSIGNMENT-3

YASH GUPTE

Date: 5th July 2019

Q1) [10 points] Read Sha, Rajkumar, et al paper, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization" and summarize 3 main key points the paper makes. Read my summary paper on the topic as well. Finally, read the positions of Linux Torvalds as described by Jonathan Corbet and Ingo Molnar and Thomas Gleixner on this topic as well. Take a position on this topic yourself and write at least one well supported paragraph or more to defend your position based on what we have learned in class. Does the PI-futex (Futex, Futexes are Tricky) that is described by Ingo Molnar provide safe and accurate protection from un-bounded priority inversion as described in the paper? If not, what is different about it?

Summary:

The paper describes the issue of Priority Inversion and the starvation of higher priority tasks in Real Time Embedded Systems and ways to alleviate this issue. This is followed by explanation Basic Priority Inheritance protocol to alleviate this problem and then its drawbacks. The paper then introduces Priority Ceiling Protocol to remove the drawbacks of Basic Inheritance Protocol.

1. **Unbounded Priority Inversion** – The process of a higher priority task waiting on a resource held by a lower priority task is called Priority Inversion. When this wait is indeterministic, it is called Unbounded Priority Inversion. The paper discusses the cases where this scenario could occur and what issues it causes.
2. **Basic Inheritance Protocol** - The main concept of this protocol is to temporarily raise the priority of the low priority task holding a resource to the priority of the highest priority task which shares the resource. However, it has its own drawbacks which include, **direct blocking** and **push through blocking**.
3. **Priority Ceiling Protocol** – In addition to raising the priority of lower tasks, this protocol increases the priority of the “semaphore” which is used in the critical section of the task to the highest priority task which uses the semaphore during the duration of its critical section. This maximum priority of the semaphore is called its “ceiling”. The paper then describes this protocol, helps resolve deadlock and chain blocking. However, this introduces another type of blocking called “ceiling blocking”. This blocking has lower overhead as compared to the other 2 blocking mechanisms.

Linus Torvalds vs Ingo Molnar Discussion:

The argument stems from an email exchanged between Ingo Molnar and Linus Torvalds regarding the introduction of Priority Inheritance (PI) based futexes for solving unbounded priority inversion issues with Real-Time Linux. Ingo Molnar wanted to introduce this feature via a kernel patch for Linux users. While Linus Torvalds had his reservations on this idea. He stated the following in his mail, "**Friends don't let friends use priority inheritance**". **Just don't do it. If you really need it, your system is broken anyway.** But Ingo Molnar went ahead and introduced a Priority Inheritance based futex patch for Linux. The specifications of the futex have been listed in his document.

Based upon the reading of the paper and arguments of both the parties, I have decided to support the notion that Ingo Molnar makes while also agreeing that in certain situations, PI may not help alleviate the situation. The following are my arguments:

1. Unbounded priority inversion is a common scenario which occurs in scheduling Real-Time systems on any platform. The only way to ensure that such a scenario does not occur at all is by using a Cyclic scheduling mechanism where each task is processed at regular intervals and resources are released after each task completes its execution. However, this method has its drawbacks when it comes to hard-realtime systems. Lower priority tasks can often get neglected due to the high rate of occurrence of high priority tasks. In most other scheduling mechanisms blocking of tasks occurs. Which in-turn causes priority inversion, which could lead to unbounded priority inversion.

2. Futexes are not present without drawbacks. They face the same issues which are faced by Basic Priority Inheritance protocol, which include push through blocking and direct blocking.
3. However, the absence of PI-Futexes does not solve the existing problems at all. The presence of minor overheads can be over-looked when it comes to not being able to schedule tasks at all.

Reasoning of FUTEX:

Advantages:

Futexes are mechanism used to help facilitate lock-based systems. Locked systems include multi-process systems which share resources between each other and prevent contention via locks. PI-Futex claims to not use kernel space at all, making the process completely user-space based. The user space operations to lock and unlock futexes are atomic which ensures proper locking.

Disadvantages:

PI- futexes' face the same issues as with Priority Inversion in any embedded system.

Taking an instance from Jonathan Corbett's article, *"When there is contention, instead, the FUTEX LOCK PI operation is requested from the kernel. The requesting process is put into a special queue, and, if necessary, that process lends its priority to the process actually 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."*

This mentions the issue of **chained priority inheritance**. Priority Inheritance works in the following manner.

In this, a lower priority task acquires a resource S1. Now a higher priority task H wants to acquire the resource. But since the resource is locked by L, H gets blocked. In order to resolve this, H temporarily lends its priority to L, so that 'L' can finish execution and let H resume its execution.

The issue: When L is executing, and another task H+1 having a higher priority than H, preempts L and tries to acquire S1. It also gets, blocked the same way as H. Now, H+1 lends its priority to L for completion. Once L has released S1, H+1 acquires it and executes its critical section. This is followed by H. The same process can continue for H+n tasks. Thus, increasing complexity and unbounded inversion times.

Q2. [15 points] Review the terminology and describe clearly what it means to write "thread safe" functions that are "re-entrant". There are generally three main ways to do this: 1) pure functions that use only stack and have no global memory, 2) functions which use thread indexed global data, and 3) functions which use shared memory global data, but synchronize access to it using a MUTEX semaphore critical section wrapper. Describe each method and how you would code it and how it would impact real-time threads/tasks. Now, using a MUTEX, provide an example using RT-Linux Pthreads that does a thread safe update of a complex state with a timestamp (pthread_mutex_lock). Your code should include two threads and one should update a timespec structure contained in a structure that includes a double precision attitude state of {X,Y,Z acceleration and Roll, Pitch, Yaw rates at Sample_Time} (just make up values for the navigational state and see http://linux.die.net/man/3/clock_gettime for how to get a precision timestamp). The second thread should read the times-stamped state without the possibility of data corruption (partial update).

Pure functions that use only stack and have no global memory

These can also be called re-entrant functions.

Method: Each thread must operate only on local variables i.e. variables stored on stack of that function. This ensures that even the control switches to another thread or ISR, the variables in the original thread are not affected.

Description with code:

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

```
pthread_t pthread_1, pthread_2;
```

```
void* callback_pthread_1(void)
{
    int variable_1=5;
    printf("Variable-1 is : %d\n",variable_1);
    return NULL;
}
```

```
void* callback_pthread_2(void)
{
    int variable_1=1;
    printf("Variable-1 is : %d\n",variable_1);
    return NULL;
}
```

```
int main(void)
{
    int i = 0;
    int err;
```

```
    if(pthread_create(&pthread_1, NULL, &callback_pthread_1, NULL))printf("Thread-1 not
created\n");
    if(pthread_create(&pthread_2, NULL, &callback_pthread_2, NULL))printf("Thread-2 not
created\n");
```

```

pthread_join(pthread_1,NULL);
pthread_join(pthread_2,NULL);

return 0;
}

```

```

Variable-1 is : 1
Variable-1 is : 5

```

In the above code even if any one of the threads is preempted by any other thread, the operation of the thread does not get affected even though the variable name is the same. This is because each of the instances of ***variable_1*** is stored on its callback function's stack.

Impact on Real-Time systems:

In real-time systems, it is necessary to share variables between processes. With this implementation, the variables to be shared will be passed as parameters to the callback functions. The processed data may need to be returned from the callback functions as well. The advantage of this method is that even if a thread is preempted midway by another thread or due to an ISR, the variables being operated on are not affected. The drawback of this method is that, a lot of stack space is used by each thread in the system. In a large real-time system with limited resources especially memory will need to be large to store not only the variables local to each thread but also during context switch.

2. Functions which use thread indexed global data

Method: Each thread stores data globally in a variable but individual instance of that variable is accessible by all threads separately without affecting the variable in other threads.

```

#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
#include <time.h>
#include <sys/time.h>

#define SIZE    (2)

int global_index_data[2]={0x24,0x45};
pthread_t threads[3];

void* (*function_ptr[3])(void);

void* read_data(void)
{
    const int* ptr = global_index_data;
    /*ptr+=1;    //should fail
    static int i=0;
    for(i=0;i<SIZE;i+=1)
        printf("Data Read:0x%x\n",ptr[i]);

```

```

        return NULL;
    }

void* thread_1(void)
{
    int* const ptr = &global_index_data[0];
    *ptr = 0xFF;
    printf("Thread-1 modified data\n");
    return NULL;
}

void* thread_2(void)
{
    int* const ptr = &global_index_data[1];
    *ptr = 0xAA;
    printf("Thread-2 modified data\n");
    return NULL;
}

int main(void)
{
    static int i=0;

    function_ptr[0] = thread_1;
    function_ptr[1] = thread_2;
    function_ptr[2] = read_data;

    for(i=0;i<2;i+=1)
        printf("Data original: 0x%x\n",global_index_data[i]);

    for(i=0;i<3;i+=1)
    {
        pthread_create(&threads,NULL,function_ptr[i],0);
        usleep(10000);
    }

    for(i=0;i<3;i+=1)
    {
        pthread_join(threads[i],NULL);
    }

    return 0;
}

```

```

Data original: 24
Data original: 45
Thread-1 modified data
Thread-2 modified data
Data Read:ff
Data Read:aa

```

In the code above, thread_1 and thread_2 write to the global_index_array[] elements. Read_data() reads this information. It is ensured by using constant pointers to integer that the value of the data in global_index_array can be altered but not the pointer itself. In case of read_data, a pointer to integer constant is used which insures that the value can be read but not modified. After running the code, the output displays the modified values of 0xFF and 0xAA by thread_1 and thread_2.

Impact on Real-Time systems:

In the use of thread indexed global data, a global array is defined with each index corresponding to a unique data belonging to a separate thread. For example, in the code above, in array global_index_data[2], index 0 corresponds to data provided by threads[0] and index 1 corresponds to data provided by threads[1]. This allows simultaneous update of data without issues of data corruption.

In Real-Time systems thread indexed global data provides a couple of advantages.

1. Firstly, data protected from corruption. In multithread applications threads can preempt each other and during data copies a variable can be modified before a write or read which can cause data corruption. Using a common global array with each index only accessed by a single thread, the possibility of data corruption is eliminated.
2. Secondly, it can be ensured that a pointer to integer constant will be able to read the data but not modify its contents.

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

Method: Use of mutexes/semaphores enables threads to operate on data i.e. read/write without causing alteration of data by any other thread/ISR. This does not prevent the preemption or context switch but protects the data thus allowing other threads/processes to operate on data not protected by mutex/semaphores.

Description and Code:

```

/*****
*****
* Code reference taken from : 1. https:linux.die.net/man/3/pthread_mutex_lock
*
* 2. https:linux.die.net/man/3/clock_gettime
*
* 3. https:www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/
*
* 4. http:www.cs.tufts.edu/comp/111/examples/Time/clock_gettime.c
*****/

```

```

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

```

```

#define MUTEX_PRESENT (1)

```

```

pthread_t pthread_arr[2];
pthread_mutex_t MUTEX;

struct DATA_STRUCT* data_struct_read;

int common_variable, update_status;

void* (*function_ptr[2])(void);

void* CallBack1(void)
{
    #if (MUTEX_PRESENT == 1)
    pthread_mutex_lock(&MUTEX);
    #endif
    sleep(3);
    common_variable+=1;
    #if (MUTEX_PRESENT ==1)
    pthread_mutex_unlock(&MUTEX);
    #endif

    return NULL;
}

void* CallBack2(void)
{
    #if (MUTEX_PRESENT == 1)
    pthread_mutex_lock(&MUTEX);
    #endif
    printf("Updated variable:%d\n",common_variable);
    #if (MUTEX_PRESENT ==1)
    pthread_mutex_unlock(&MUTEX);
    #endif

    return NULL;
}

int main(void)
{
    int i = 0;
    int error;

    function_ptr[0]=CallBack1;
    function_ptr[1]=CallBack2;

    common_variable=0;
    update_status=0;

    if (pthread_mutex_init(&MUTEX, NULL))
        printf("Mutex initialization failed\n");
}

```



```

for(i=0;i<2;i+=1)
{
    pthread_create(&pthread_arr,NULL,function_ptr[i],0);
    sleep(1);
}

for(i=0;i<2;i+=1)
{
    pthread_join(pthread_arr[i],0);
}

pthread_mutex_destroy(&MUTEX);

return 0;
}

```

```

Without Mutex:
Updated variable:0

With Mutex:
Updated variable:1

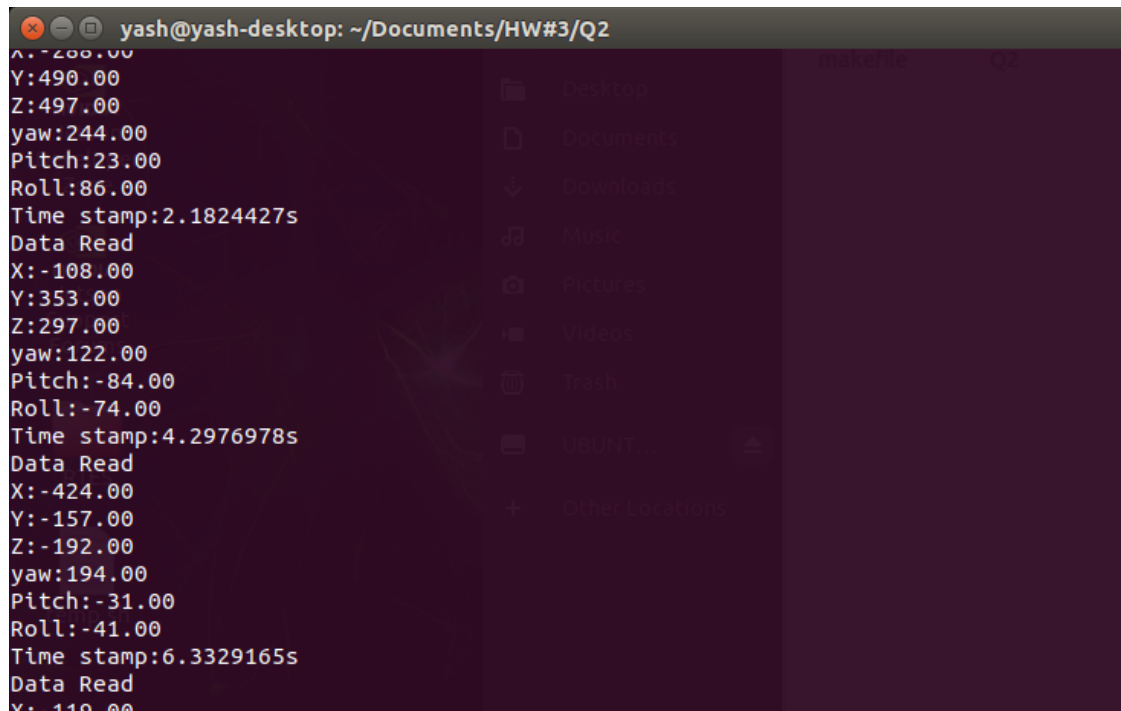
```

In the code, a mutex called MUTEX protects the update and display of “common_variable”. When the code is run with `#define MUTEX_PRESENT (1)`, the mutex is implemented. When that happens, the `pthread_mutex_lock` and `pthread_mutex_unlock` is used to protect data during update and display. This ensures that the displayed value is always updated. On the other hand, when `#define MUTEX_PRESENT (0)`, mutex locks and unlocks are not implemented and the output is unpredictable in the sense that the output may or may not be updated because there is no mechanism to stop print of the variable before the update.

Implementation in Real-Time Systems:

In Real-Time systems, sharing data between threads is necessary. Mutexes allow for sharing global data between threads without the chance of data read during the update process and vice-versa. This is a very important feature of mutexes. This removes the need to have multiple copies of variables on the function’s stack. There are however other problems with mutexes mainly “**deadlocks**”. In deadlocks, one task which acquires a mutex M2 is preempted by another task which already acquired M1. This task now tries to lock M2 but gets blocked because M2 is with the other task. Task with M2 during its execution needs M1 which is with another task. As a result, both tasks enter a deadlock. Mutexes also increase overhead in total task execution time.

Code for the Mutex implementation with timestamp is attached separately.

A terminal window titled 'yash@yash-desktop: ~/Documents/HW#3/Q2' displays sensor data. The data is organized into three blocks, each starting with 'Data Read'. The first block shows X: -108.00, Y: 353.00, Z: 297.00, yaw: 122.00, Pitch: -84.00, Roll: -74.00, and Time stamp: 4.2976978s. The second block shows X: -424.00, Y: -157.00, Z: -192.00, yaw: 194.00, Pitch: -31.00, Roll: -41.00, and Time stamp: 6.3329165s. The third block is partially visible. To the right of the terminal is a file manager sidebar with icons for Desktop, Documents, Downloads, Music, Pictures, Videos, Trash, UBUNTU, and Other Locations. The main area of the file manager is dark and mostly empty.

```
x: -200.00
Y: 490.00
Z: 497.00
yaw: 244.00
Pitch: 23.00
Roll: 86.00
Time stamp: 2.1824427s
Data Read
X: -108.00
Y: 353.00
Z: 297.00
yaw: 122.00
Pitch: -84.00
Roll: -74.00
Time stamp: 4.2976978s
Data Read
X: -424.00
Y: -157.00
Z: -192.00
yaw: 194.00
Pitch: -31.00
Roll: -41.00
Time stamp: 6.3329165s
Data Read
x: -110.00
```

In the code, MUTEX is a mutex variable used preventing over-writing of data. Function UpdateTime first locks the MUTEX then updates the values of X, Y & Z and timestamp then unlocks the MUTEX. Function GetTime() first locks MUTEX then this function reads this data and displays it. This is followed by unlocking the MUTEX. If during the UpdateTime() or GetTime() one task tries to preempt the other task, it is blocked till the mutex is released.

Q3) [15 points] Download example-sync/ and describe both the issues of deadlock and unbounded priority inversion and the root cause for both in the example code. Fix the deadlock so that it does not occur by using a random back-off scheme to resolve. For the unbounded inversion, is there a real fix in Linux – if not, why not? What about a patch for the Linux kernel? For example, Linux Kernel.org recommends the RT_PREEMPT Patch, but would this really help? Read about the patch and describe why think it would or would not help with unbounded priority inversion. Based on inversion, does it make sense to simply switch to an RTOS and not use Linux at all for both HRT and SRT services?

Deadlock:

```
yash@yash-desktop: ~/Documents/HW#3/Q3_deadlock
cc -O -g -DLINUX -o Q3_deadlock Q3_deadlock.o -lpthread -lrt
make: warning: Clock skew detected. Your build may be incomplete.
yash@yash-desktop:~/Documents/HW#3/Q3_deadlock$ ./Q3_deadlock
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
THREAD 1 grabbing resources
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: b2c811f0 done
THREAD 2 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
Thread 2: b24801f0 done
mutex A destroy: Success
mutex B destroy: Success
All done
yash@yash-desktop:~/Documents/HW#3/Q3_deadlock$
```

Task A and Task-B one gets a resource and the other does not. Vice versa causing a deadlock.

Solution:

A custom function ReduceBackOff () is used to solve the issue of Deadlock. It is based off reduce back-off time. Reduce Back-Off time states that, when a deadlock arises, one of the services must be restarted in with a random time difference between them to allow each service to lock and unlock the shared resources without competing for it.

This function removes the deadlock. It is invoked from either thread when a deadlock is detected.

Resource-A obtained by thread[0] is released so that thread[1] can complete execution the same happens when thread[1] attains Resource-B. After which that resource is locked again. Since, thread[1] has now finished execution, Mutex-B need not be locked.

Once, it detects that a deadlock occurred, it performs the following steps:

1. It unlocks the mutex which the current thread just locked.
2. It puts the CPU to sleep for a random amount of time. (Max. up to 10000us)
3. It locks the mutex again.

This allows the other thread to lock this semaphore, since it already has 1 semaphore, it can

Now lock both the mutexes and complete its critical section during the time the `usleep()` function is called. It then releases both the mutexes. Now the current thread can obtain both mutexes without any risk of deadlocks.

Unbounded Priority Inversion:

```
root@yash-desktop: ~/Downloads/Qns/Q3_unbounded
root@yash-desktop:~/Downloads/Qns/Q3_unbounded# ./Q3_unbounded
Usage: pthread interfere-seconds
root@yash-desktop:~/Downloads/Qns/Q3_unbounded# ./Q3_unbounded 3
interference time = 3 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 1562275369 sec, 212669 nsec
Creating thread 2
Middle prio 2 thread spawned at 1562275370 sec, 212819 nsec
Creating thread 1, CSnt=1
High prio 1 thread spawned at 1562275370 sec, 212878 nsec
**** 2 idle NO SEM stopping at 1562275370 sec, 214154 nsec
**** 3 idle stopping at 1562275371 sec, 215370 nsec
LOW PRIO done
MID PRIO done
**** 1 idle stopping at 1562275373 sec, 218100 nsec
HIGH PRIO done
START SERVICE done
All done
root@yash-desktop:~/Downloads/Qns/Q3_unbounded#
```

In the code, High and Low priority tasks share a mutex. Medium priority task does not depend on the mutex. All 3 threads have a common task of calculating Fibonacci series.

The code initializes the tasks in ascending order.

- i. Low priority task, which locks the mutex and executes.
- ii. Medium priority task preempts low priority task and executes.
- iii. High priority task is initiated which needs the mutex present with low priority task but Medium priority task is executing currently hence High priority task does not get chance to execute and gets blocked.

Low -> initialized at 212669ns – takes semaphore

Med -> init @ 212819ns – preempts Low task and executes

High -> init @ 212878ns – preempts med task and tries to take sem from low task but fails and gets blocked.

Med finishes execution at 214154ns blocking high task for $(212878 - 214154)\text{ns} = 1276\text{ns}$.

Low then finishes execution at 215370ns.

After it releases the semaphore, High task acquires it and finishes execution at 218100ns.

This means approximately, 1276ns High task was blocked due to unbounded priority inversion.

PREEMPT_RT patch:

RT_PREEMPT patch: <https://stackoverflow.com/questions/26311757/what-is-rt-preempt-how-is-it-different-from-preempt-rt-does-these-mean-same-re>

Link2: <https://lwn.net/Articles/146861/>

In PREEMT_RT patch, there is a special provision to introduce priority inheritance into the kernel. With the specifications described by Paul McKenny, in his article “A realtime preemption overview” he states various features of the patch as further elaborated by Steven Rostedt in his 2013 conference.

This PI-based kernel introduced the capability to boost the priority of lower priority task containing a locked semaphore to the priority of the highest task that could preempt the lower task. **This feature can alleviate the situation which arose in the unbounded priority inversion question.** The solution can be described as follows:

1. Low priority task acquires the lock on S1 and gets the priority of High task since it shares semaphore S1 with High task.
2. Medium task tried to preempt the low priority task but fails because the lower task has the same priority as the high priority task.
3. Low task finishes execution, releases semaphore and returns to its original low priority. High task then executes by locking the semaphore and performing its critical section.
4. This is followed by medium priority task.

However, the patch does not come without drawbacks. The PI is transitive i.e. a higher priority task than High let's say H2 which does not require S1 can preempt low priority task. This can further be preempted by another higher priority task H3 etc.

RTOS vs Linux:

Link: <https://www.electronicsworld.com/market-sectors/embedded-systems/analysis-linux-versus-rtos-2008-08/>

1. In traditional linux, real-time operations were achieved by using a separate real-time core with a separate scheduler. It is possible to use this “micro-kernel” for hard real time systems.
2. RTOS have the entire system dedicated to real-time operations, which means the memory, CPU, cache, registers etc have been selected and operated in a manner to completely support real-time operations. With the existence of a separate real-time core, in Linux, real-time services can be provided but there is a crunch on the limited resources which are available with the real-time linux. The memory, buses, cache etc will need to be shared with the traditional core of Linux which has been designed for best-effort applications.
3. With the introduction of patches like PI-futexes and PREEMT_RT kernel, there have been significant upgrades to utilization of resources for real-time systems. However, these are still in the nascent stage and need to be tried and tested for them to become completely reliable.
4. All in all, mainstream Linux is useful for soft real time systems and for hard-real time systems which do not require large amount of resources.

- Q4. [15 points] Review `heap_mq.c` and `posix_mq.c`. First, re-write the VxWorks code so that it uses RT-Linux Pthreads (FIFO) instead of VxWorks tasks, and then write a brief paragraph describing how these two message queue applications are similar and how they are different. You may find the following Linux POSIX demo code useful, but make sure you not only read and port the code, but that you build it, load it, and execute it to make sure you understand how both applications work and prove that you got the POSIX message queue features working in Linux on your Jetson. Message queues are often suggested as a way to avoid MUTEX priority inversion. Would you agree that this really circumvents the problem of unbounded inversion? If so why, if not, why not?
- Heap:**

```
root@yash-desktop: ~/Downloads/Qns/Q4_heap
root@yash-desktop:~/Downloads/Qns/Q4_heap# ls
makefile  Q4_heap_1  Q4_heap_1.c  Q4_heap_1.d  Q4_heap_1.o
root@yash-desktop:~/Downloads/Qns/Q4_heap# ./Q4_heap_1
buffer =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~System has 4 processors configured and 4 available.
Using CPUS=1 from total available.
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=30
rt_min_prio=1
Service threads will run on 1 CPU cores
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
send: message ptr 0x8C000B20 successfully sent
Reading 8 bytes
receive: ptr msg 0x8C000B20 received with priority = 30, length = 12, id = 999
contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
heap space memory freed

TEST COMPLETE
root@yash-desktop:~/Downloads/Qns/Q4_heap#
```

[Screenshot]

The main loop generates two threads, sender and receiver.

Sender Thread:

The sender thread mallocs a pointer of size buffer which is a global variable. This pointer is loaded onto the message queue to send to receiver.

Receiver Thread:

The receiver reads the queue and receives the most recent message since the scheme is FIFO. Once received the memory is freed.

POSIX:

The main loop generates two threads, sender and receiver.

```
root@yash-desktop: ~/Downloads/Qns/Q4_posix
mq_close(SNDRCV_MQ);
~~~~~
In file included from Q4_posix_1.c:34:0:
/usr/include/mqueue.h:43:12: note: expected 'mqd_t {aka int}' but argument is of type 'char *'
extern int mq_close (mqd_t __mqdes) __THROW;
~~~~~
cc -O -g -D_LINUX -o Q4_posix_1 Q4_posix_1.o -lpthread -lrt
root@yash-desktop:~/Downloads/Qns/Q4_posix# ./Q4_posix_1
System has 4 processors configured and 4 available.
Using CPUS=1 from total available.
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=30
rt_min_prio=1
Service threads will run on 1 CPU cores
sender opened mq
send: message successfully sent
receive: 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

TEST COMPLETE
root@yash-desktop:~/Downloads/Qns/Q4_posix#
```

[Screenshot]

Sender Thread:

The sender thread directly sends `canned_msg` via the queue.

Receiver Thread:

The receiver copies the first message on the queue into a local buffer in order to read the data.

Heap vs Posix:

Differences:

1. Posix queues pass variables containing data directly, whereas, heap first mallocs space for the same variable pointer then passes the pointer in the calling function.
2. Due to passing of a pointer via malloc, more memory is occupied by the Heap method. This method of mallocing memory is implemented in both send and receive.

Similarities:

1. Heap as well as Posix queues, utilize global data to send information in send and receive functions. In case of Heap, the global data is copied into a pointer whereas in Posix, the global data is copied into a local variable which is then sent into the queue.
2. The underlying mechanism of exchanging information is via message queues in both the implementations. In Linux, they make use of the “`mqueue.h`” header for utilizing the queue features.

Message Queues properties:

POSIX message queues are quite robust in that they can be used to protect global data as well avoid unbounded priority inversion in real-time systems. The following are the features which make message queues good candidates for implementation in Real-Time Embedded systems.

1. Message queues do not Block:

Communication between processes is faster since, no process waits for its data to be written or read by another process. This feature belongs to `MQ_NONBLOCK` feature of POSIX message queues. In this implementation, `mq_send()` or `mq_receive()` never block, if no data is available an “`errno`” set to `EAGAIN`.

2. Message queues can boost priority:

Normally every message which is sent via `write()` has the lowest priority of ‘0’. If `MQ_PRIO_BOOST` is implemented, whenever a ‘0’ priority message is written, its priority is boosted up-to that of the calling process. This behavior is analogous to a certain extent with Priority Inheritance in which the semaphores are given the priority of the highest preemptible task with the shared semaphore. Using `MQ_PRIO_BOOST` can prevent preemption of the message by the same or lower priority task.

3. Message queues can give preference to certain tasks arbitrarily:

Another feature which can allow for prevention of unbounded priority inversion is `MQ_PRIO_RESTRICT`. In this option, any process using the `mq_send()` can ensure that no task with a higher priority can preempt the current tasks resource.

4. Message queues are asynchronous:

POSIX message queues allow processes to run independently from the queues. This means that in case a new process is added or removed, they can still utilize the message queues. This allows a process with multiple named queues to be operated on by variety of processes overtime.

Q5. [30 points] Watchdog timers, timeouts and timer services – First, read this overview of the Linux Watchdog Daemon and describe how it might be used if software caused an indefinite deadlock. Next, to explore timeouts, use your code from #2 and create a thread that waits on a MUTEX semaphore for up to 10 seconds and then un-blocks and prints out “No new data available at ” 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.

Linux Watchdog Daemon:

1. Most embedded systems have a watchdog device which is responsible for a timeout causing a system restart. It is mainly useful in scenarios where the system hangs indefinitely. This could be because it ran out of memory, tried accessing a null pointer, entered a non-terminating loop etc. Watchdog daemon is a software which runs in the background of Linux machines and periodically refreshes the watchdog devices before it expires to prevent reboot.
2. The watchdog daemon's repeated refresh procedure could in some cases prevent the automatic recovery of the system. For example, if a process is waiting to access a block of memory but is left waiting indefinitely due to a pointer error. The daemon task would repeatedly refresh the watchdog device causing the process stalling to persist.
3. To prevent this kind of a situation, it performs a set of **three basic tests** to ensure that the system is functioning normally. These tests include checking for **resources available** – memory, checking for the **execution of expected processes** and verifying if **resources needed for executing** certain processes are available or not.
4. If either of these tests fail, the daemon can reboot the system using a “moderate” approach while maintaining a log of what caused the error. This “moderate” shutdown is not the conventional shut-down of systems since traditional approach will try to access resources before shutting down and a result in contention leading to prevention of shutdown or unexpected behavior.
5. The “moderate” shutdown approach is called “blunderbuss approach”. In this process, firstly a SIGTERM is sent to all processes followed by SIGKILL after a gap of 5s to ensure all processes are suspended. The daemon records the shutdown, updates the random seed and synchronizes the CMOS clock to system clock to ensure when the system restarts, the system time is maintained. Daemon task also synchronizes and un-mounts file systems to prevent corruption at reset.
6. There are 2 main types of watchdog daemons, ‘wd_keepalive’ and ‘watchdog’. ‘Wd_keepalive’ provides the hardware driver open, close and refresh operations whereas ‘watchdog’ provides system check operations as well. The combination of these 2 daemons is used to successfully operate the “moderate” shutdown as well as power on.
7. At boot, ‘wd_keepalive’ is initiated since it can handle faults which could be encountered at boot time. Once all systems are up and running, ‘watchdog’ is initiated to carry out the above 3 tests. Similarly, at shutdown, first ‘watchdog’ is stopped followed by ‘wd_keepalive’ since it can handle faults at shutdown.

Application with Deadlocks:

1. When a deadlock is detected, it is mostly an occurrence which goes undetected during the normal testing because it may be caused very rarely. This could mainly be because a process tried to acquire a resource at the very same instant at which another process was using it and the processes were designed to not ever get processed at the same instant. However, due to a clock skew or missed deadlines, the task happened to interact. In this situation, the ‘watchdog’ daemon will perform the 3 tests which will fail. As a result, the “moderate” shutdown approach will be used. The SIGKILL will ensure that the processes causing the deadlock to terminate.
2. On reboot, all resources will belong with the designated processes. This gives a chance for the process to run again and the chance of the clock-skew occurring at the same instant is very low.

Adaptation of Code#2:

```
yash@yash-desktop: ~/Documents/HW#3/Q5
yash@yash-desktop:~/Documents/HW#3/Q5$ ./Q5
Mutex acquired at 1562276340.723447369
Data Written
No new data available at 1562276350.723999865
No new data available at 1562276360.724818924
Unlocked
X:-414.00
Y:206.00
Z:-180.00
yaw:253.00
Pitch:51.00
Roll:44.00
Time stamp:0.652083
Mutex acquired at 1562276380.724698603
Data Written
No new data available at 1562276380.727323083
No new data available at 1562276390.728198183
No new data available at 1562276400.728977658
Unlocked
X:-296.00
Y:-458.00
Z:-172.00
yaw:18.00
Pitch:-28.00
```

[Code attached separately]

In the code the functions `TimedAccess ()` and `UpdateTime ()` have been modified to incorporate `timed_mutex` locks. The function `UpdateTime ()` stores the data in `data_struct_data` like Q2. In `TimedAccess ()` a Continuous loop checks if timeout of 10s has elapsed. If it has, it prints "No new data available at <time>". If it hasn't i.e. it has acquired the mutex, the read structure is populated.

In `UpdateTime ()` Mutex is locked first then function populates the `write_data_structre` with X, Y, Z and time values. the mutex is freed after these operations. `Sleep ()` is used to display functionality of `timed_mutex`.