

### Exercise 3

### Real- time Embedded systems (ECEN 5623)

Note : For this exercise the Jetson TK1 board was used

#### **Problem-1**

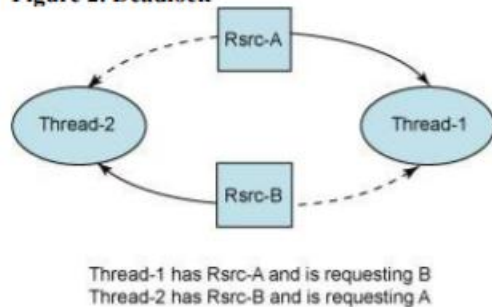
In real time systems when resources are being shared by more than one thread of execution, certain problems are encountered. These problems are :

- Resource Deadlock
- · Unbounded Priority Inversion

#### **Resource Dead lock:**

Deadlock is a scenario that occurs when a thread occupies a resource that is requested by another thread and the resource needed by this thread is occupied by the other thread. In the figure shown below, thread 1 has resource A and is requesting resource B, Thread 2 has resource B and is requesting resource A. thus a circular wait state occurs, which arises from the shared resources used in common between 2 threads.

**Figure 2. Deadlock**



3 common methods used to recover the systems from deadlock are:

- Avoidance: In this method, to avoid circular wait states the acquisition of multiple resources is carried out in a serialized manner
- Using random back-off: This method involves a backoff time which is used to delay the acquisition of resources to prevent multiple access of the same resource which might result in deadlock.
- Using hardware watchdog: When a deadlock occurs and the system is stuck then, a watchdog timer can be used to detect this and reboot the system to start the recovery.

#### **Unbounded Priority Inversion:**

Consider three services, A higher priority service H, medium priority service M and a lower priority service L. if a lower priority service and higher priority service share a resource and if

the resource is acquired by the lower priority service, then if the higher priority service H it is blocked by the service L. while L is executing in the critical section, if one or more services with medium priority interferes then this service will be executed first and until the medium priority services occur and get executed the higher priority service will not run. Thus the priority inversion takes place for an unbounded time.

2 Solutions have been used to deal with the priority inversion problem:

- Priority Inheritance Protocol
- Priority ceiling protocol

The above 2 solutions to the priority inversion problem have been described in detail in the paper

**The key points that are stated by the paper are as follows:**

- Priority inversion occurs when a higher priority job is blocked by a lower priority job. This arises when 2 jobs attempt to access a shared resource. If the access of the resources is in a serial manner then this can be avoided. Also, if the access to the shared resource is gained by a higher priority task prior to the lower priority task then the problem doesn't occur. But if a lower priority task gains access first then blocking occurs. The degree of schedulability for a system reduces with blocking. Semaphores and mutexes are used for synchronization which protect the consistency of the shared data, but sometimes they reduce the ability of system to meet its deadline.
- To solve the problem of priority inversion, a protocol called the priority inheritance protocol is used. This protocol states that when a Job blocks one or more higher priority job, it inherits the highest priority i.e. it ignores its assigned priority, inherits the highest priority, completes execution of its critical section and regains its original priority. The main key points to be noted in this protocol are
  1. A high priority job can be blocked by a lower priority job for at most the duration of 1 critical section because a higher priority task can be blocked if the lower priority task is executing in its critical section.
  2. Push through blocking can be caused by the a semaphore if it is accessed by a job that has lower priority than the current job or can inherit a priority equal to or higher than itself.
  3. If there are a number of semaphores, say m, lock a job, then this job can be locked m times.
- The priority inheritance protocol does not prevent the deadlock, also even though the blocking time is bounded, it can still be significant due to the chain of blocking that is formed due to nesting of the task and acquisition of semaphores. To prevent the problems of deadlock and chained blocking the priority ceiling protocol is used
- The priority ceiling protocol states that when a job preempts the critical section of another job and executes its own critical section, then the priority of the new critical

sections is always high than the priorities of preempted the critical sections, if this condition is not satisfied then, the new job is denied entry into the critical section and is suspended. To realize this idea, a priority ceiling is assigned to each semaphore which is equal to the highest priority task that uses this semaphore. The key points to be noted are:

1. Since the priority ceiling is given to semaphores, a task that has inherited higher priority and has blocked the higher priority task, gets all the requested semaphores and hence solves the problem of deadlock
  2. This protocol prevents transitive blocking which is said to occur if a job is blocked by a job which is blocked by another job. Hence it prevents the problem of chain blocking.
- With the use of these protocols blocking can be avoided and hence the schedulability of the system is improved.

### **Jonathan Corbet's position on priority inheritance protocol:**

According to Jonathan there are a number of approaches to avoiding priority inversion, including lockless designs, carefully thought-out locking scenarios, and a te priority inheritance. The priority inheritance is a method in which when a lock is taken by a low-priority process, the priority of that process is boosted until the lock is released. He says that priority inheritance shows a tendency to complicate and slow down the locking code and hence are not used in many applications. He is not supportive of the idea of priority inheritance.

Another developer called Igno has opposing views to this and has given a PI-Futex implementation.

### **Igno Molnar's position on priority inheritance protocol:**

Igno says that lockless a code is impossible and there are some applications where lockless access cannot be done. Igno molnar offers a solution where locking is implemented in the user space. This space can be preempted and a critical section can be preempted. He offers a solution wherein PI-Futex is used. The PI Futex is used for lock and unlock. User space uses these options to lock/unlock these mutexes without entering the kernel. If a lock fails then the FUTEX\_LOCK\_PI is called and the remaining work is done by the kernel. the code looks up the task that owns the and attaches a 'PI state' structure to the futex-queue. The pi\_state includes an rt-mutex, which is a kernel-based synchronization object. Then the FUTEX\_WAITERS bit is set. Then the other task tries to lock the rt-mutex, on which it blocks. Once it returns, it has the mutex acquired, and it sets the futex value to its own thread ID and returns. User space has no other work to perform. if the unlock fails then FUTEX\_UNLOCK\_PI is called, and the kernel unlocks the futex on the behalf of user space. Thus Priority inheritance approach is supported by Igno molnar and uses the PI Futex implementation.

**Our position on this topic :**

We agree with Igno's view that creating a code without locks is very difficult to do and the synchronization objects like semaphores and mutexes are unavoidable and necessary for efficient code. The view of linux, which is that the priority should not be used and if you are using it then your code is already broken is not too accurate and cannot work all the time since in some applications usage of locks is necessary and cannot be avoided.

Priority inheritance is rather a useful approach. The implementation of PI-Futex is a good way to handle the priority inheritance since no work is done at the kernel level. Futexes are fast locking mechanism used in user space. The locking and unlocking is implemented at the user level. PI futex works faster because no kernel level calls are involved in the case where there is no contention . The kernel only does work when the lock fails by calling FUTEX\_LOCK\_PI and if an unlock fails it calls the FUTEX\_UNLOCK\_PI. The PI Futex uses the rt\_mutex which is similar to a regular mutex but also takes care of priority inheritance.

Further,even though linux is a commercial OS that does not involve real time fixes like the priority inheritance, the approach of PI-Futex ,if implemented in the mainline of the OS can be effective for the implementation of the priority inheritance protocol which solves the problem of priority inversion.

## **Problem-2**

**Thread Safety** refers to the ability of a function to be operated concurrently by multiple threads without corruption of critical variables. In other words, a thread safe code is one that can be executed simultaneously from multiple threads. Thread-safe functions can be called from multiple threads without contention of data.

On the contrary, the term **Reentrancy** is usually associated with functions that can be invoked multiple times before a current execution is complete. Any such invocation shouldn't affect the path taken by other invocations. Reentrant functions typically depend on local variables to accomplish execution without contention. Global and static variable are least likely to be used in such functions.

**Thread-safe functions that are reentrant** possess the ability to be concurrently operated by multiple threads safely and the integrity of such functions are maintained in critical situations like interrupts. Such functions can be interrupted and resume operation after servicing an interrupt. There would not be any corruption of data because reentrant functions does not use global variables that can be modified by interrupts. Also, since everything is local variable dependent and as local variables are stored in the stack, contention is least likely.

It is also worth a mention that thread-safe functions do not guarantee reentrancy by most times reentrant functions are thread-safe.

A swap function is implemented using the three methods mentioned in the question for thread safety and reentrancy. Below are the implementations using those three methods: All codes have also been uploaded to the canvas.

### **Description of methods and impact to real time systems:**

**Method-1:** Pure functions that only use stack and have no global memory. Below is the code that exhibits this method.

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>
#include<sys/syscall.h>

typedef struct my_thread
{
    int a;
    int b;
```

```
}thread_t;

void swap(int *a, int *b)
{
    int c = *a;
    *a = *b;
    *b = c;
}

void task(void *requesting_thread)
{
    thread_t * req_thread = (thread_t*) requesting_thread;
    int a,b;

    a = req_thread->a;
    b = req_thread->b;

    printf("Thread %ld - Before swap: a = %d, b = %d\n",syscall(__NR_gettid),a,b);
    swap(&a,&b);
    printf("Thread %ld - After swap: a = %d, b = %d\n",syscall(__NR_gettid),a,b);
}

int main()
{
    pthread_t thread1, thread2;
    int thread_status = 0;
    thread_t *t_child1 = NULL;
    thread_t *t_child2 = NULL;

    t_child1 = (thread_t*)malloc(sizeof(thread_t));
    if(t_child1 == NULL)
    {
        printf("\nMalloc for Child Thread-1 failed. Exiting\n");
        return 0;
    }
    t_child1->a = 2;
    t_child1->b = 8;

    t_child2 = (thread_t*)malloc(sizeof(thread_t));
    if(t_child2 == NULL)
    {
        printf("\nMalloc for Child Thread-2 failed. Exiting\n");
        free(t_child1);
        return 0;
    }
    t_child2->a = 5;
    t_child2->b = 9;

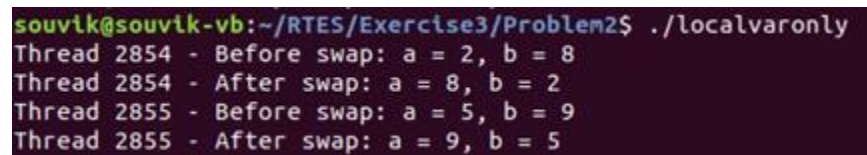
    thread_status = pthread_create(&thread1,NULL,(void *)task, (void*) t_child1);
    if(thread_status != 0)
    {
        printf("\nChild thread - 1 cannot be created\n");
        return 0;
    }
}
```

```
thread_status = pthread_create(&thread2, NULL, (void *)task, (void*) t_child2);
if(thread_status != 0)
{
    printf("\nChild thread - 2 cannot be created\n");
    return 0;
}

pthread_join(thread1, NULL);
pthread_join(thread2, NULL);

return 0;
}
```

Figure 1 shows the result of execution using method-1.



```
souvik@souvik-vb:~/RTES/Exercise3/Problem2$ ./localvaronly
Thread 2854 - Before swap: a = 2, b = 8
Thread 2854 - After swap: a = 8, b = 2
Thread 2855 - Before swap: a = 5, b = 9
Thread 2855 - After swap: a = 9, b = 5
```

*Figure.1 Executing the swap function using method-1*

This method depends on the use of local variables to achieve thread safety and reentrancy. Since threads have their own individual stacks, they would be able to concurrently access this function without race conditions. In the context of real-time embedded systems, there will be context switch in terms of which thread is executing and, in the order, determined by the scheduler with default priority. We do not account for context switches when we schedule real-time services over the LCM of the service periods. There can be scenarios where margins can be 100% and although they might be schedulable when context switches are less. It would be imperative that some services would miss their deadlines once the number of threads accessing a function increases, thereby increasing the number of context switches and hence overhead.

**Method-2:** Functions which use thread indexed global data. Below is the code that exhibits this method.

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>
#include<sys/syscall.h>

typedef struct my_thread
{
    int a;
    int b;
```

```
}thread_t;

__thread int c;

void swap(int *a, int *b)
{
    c = *a;
    *a = *b;
    *b = c;
}

void task(void *requesting_thread)
{
    thread_t * req_thread = (thread_t*) requesting_thread;
    int a,b;

    a = req_thread->a;
    b = req_thread->b;

    printf("Thread %ld - Before swap: a = %d, b = %d\n",syscall(__NR_gettid),a,b);
    swap(&a,&b);
    printf("Thread %ld - After swap: a = %d, b = %d\n",syscall(__NR_gettid),a,b);
}

int main()
{
    pthread_t thread1, thread2;
    int thread_status = 0;
    thread_t *t_child1 = NULL;
    thread_t *t_child2 = NULL;

    t_child1 = (thread_t*)malloc(sizeof(thread_t));
    if(t_child1 == NULL)
    {
        printf("\nMalloc for Child Thread-1 failed. Exiting\n");
        return 0;
    }
    t_child1->a = 2;
    t_child1->b = 8;

    t_child2 = (thread_t*)malloc(sizeof(thread_t));
    if(t_child2 == NULL)
    {
        printf("\nMalloc for Child Thread-2 failed. Exiting\n");
        free(t_child1);
        return 0;
    }
    t_child2->a = 5;
    t_child2->b = 9;

    thread_status = pthread_create(&thread1,NULL,(void *)task, (void*) t_child1);
    if(thread_status != 0)
    {
        printf("\nChild thread - 1 cannot be created\n");
        return 0;
    }
}
```



```

    }

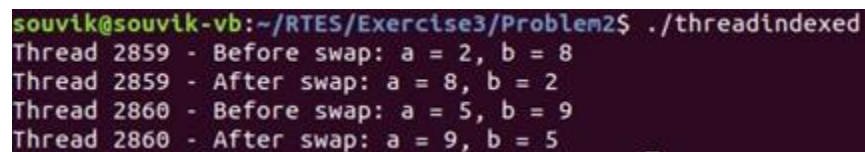
    thread_status = pthread_create(&thread2, NULL, (void *)task, (void*) t_child2);
    if(thread_status != 0)
    {
        printf("\nChild thread - 2 cannot be created\n");
        return 0;
    }

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}

```

Figure 2 shows the result of execution using method-2.



```

souvik@souvik-vb:~/RTES/Exercise3/Problem2$ ./threadindexed
Thread 2859 - Before swap: a = 2, b = 8
Thread 2859 - After swap: a = 8, b = 2
Thread 2860 - Before swap: a = 5, b = 9
Thread 2860 - After swap: a = 9, b = 5

```

*Figure.2 Executing the swap function using method-2*

This method depends on the use of thread local storage to achieve thread safety and reentrancy. When an address-of operator is applied to such a variable, they are evaluated at runtime and return the address of the current thread's instance of that variable. This address can then be used by that particular thread. The scope of this variable is only limited to the lifetime of that specific thread. When a thread terminates any pointers to the thread-local variables in that thread becomes invalid. With respect to real-time scenarios, this method is expected to work better than the previous method in terms of adding overheads to service deadline margins. But there could be reentrancy issues. Even though the threads are safe from each other, if multiple invocations to the swap function were to be made within a single thread, the value of the global variable would have been unpredictable.

**Method-3:** Functions which use shared memory global data but synchronize access to it using a MUTEX semaphore critical section wrapper. Below is the code that exhibits this method.

```

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

pthread_mutex_t resource_lock;

typedef struct my_thread
{
    int a;

```

```
        int b;

    }thread_t;

    int c;

    void swap(int *a, int *b)
    {
        pthread_mutex_lock(&resource_lock);
        c = *a;
        *a = *b;
        *b = c;
        pthread_mutex_unlock(&resource_lock);
    }

    void task(void *requesting_thread)
    {
        thread_t * req_thread = (thread_t*) requesting_thread;
        int a,b;

        a = req_thread->a;
        b = req_thread->b;

        printf("Thread %ld - Before swap: a = %d, b = %d\n",syscall(__NR_gettid),a,b);
        swap(&a,&b);
        printf("Thread %ld - After swap: a = %d, b = %d\n",syscall(__NR_gettid),a,b);
    }

    int main()
    {
        pthread_t thread1, thread2;
        int thread_status = 0;
        thread_t *t_child1 = NULL;
        thread_t *t_child2 = NULL;

        t_child1 = (thread_t*)malloc(sizeof(thread_t));
        if(t_child1 == NULL)
        {
            printf("\nMalloc for Child Thread-1 failed. Exiting\n");
            return 0;
        }
        t_child1->a = 2;
        t_child1->b = 8;

        t_child2 = (thread_t*)malloc(sizeof(thread_t));
        if(t_child2 == NULL)
        {
            printf("\nMalloc for Child Thread-2 failed. Exiting\n");
            free(t_child1);
            return 0;
        }
        t_child2->a = 5;
        t_child2->b = 9;

        if(pthread_mutex_init(&resource_lock, NULL) != 0)
```

```
{
    printf("\nMutex initialization failed. Exiting\n");
    return 0;
}

thread_status = pthread_create(&thread1, NULL, (void *)task, (void*) t_child1);
if(thread_status != 0)
{
    printf("\nChild thread - 1 cannot be created\n");
    return 0;
}

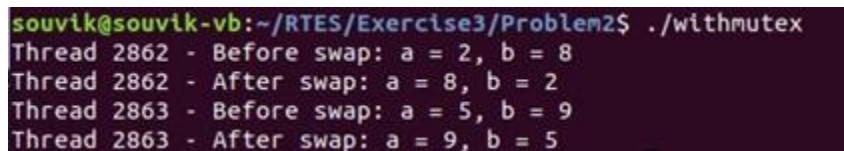
thread_status = pthread_create(&thread2, NULL, (void *)task, (void*) t_child2);
if(thread_status != 0)
{
    printf("\nChild thread - 2 cannot be created\n");
    return 0;
}

pthread_join(thread1, NULL);
pthread_join(thread2, NULL);

pthread_mutex_destroy(&resource_lock);

return 0;
}
```

Figure 3 shows the result of execution using method-3.



```
souvik@souvik-vb:~/RTES/Exercise3/Problem2$ ./withmutex
Thread 2862 - Before swap: a = 2, b = 8
Thread 2862 - After swap: a = 8, b = 2
Thread 2863 - Before swap: a = 5, b = 9
Thread 2863 - After swap: a = 9, b = 5
```

*Figure.2 Executing the swap function using method-3*

This method uses a global variable which is shared by both the threads. There would have been contentions and probably a race condition if it was not for the MUTEX used to lock the global resource. The MUTEX prevents one thread from over writing the global variable when another thread is using it. When a thread of execution encounters the mutex lock, it locks the resource. The other thread at this instant would be able to access the resource locked by the first thread. The lock can only be released or unlocked by the same thread which locked it, thereby synchronizing access to the shared resource and preempting race conditions. Locking a resource and synchronizing access to global variables using MUTEXes and semaphores are the most efficient ways to use global variable in Real Time multithreaded systems. Context switch overhead would be minimum for this method.

Following the program where one thread calculates pitch, roll, yaw and records timestamp of calculation while the other thread reads it without corruption. The same code has also been uploaded to canvas as an attachment.

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<time.h>
#include<math.h>
#include<unistd.h>
#include<sys/time.h>
#include<sys/syscall.h>

#define PI (3.14)

pthread_mutex_t resource_lock;

typedef struct
{
    int id;
    struct timespec    timestamp;
    double acc_x;
    double acc_y;
    double acc_z;
    double roll;
    double pitch;
    double yaw;
}obj_t;

obj_t complex_obj;

void calculate_data(void *requesting_thread)
{
    obj_t *req_thread = (obj_t*) requesting_thread;
    pthread_mutex_lock(&resource_lock);

    complex_obj.acc_x = req_thread->acc_x;
    complex_obj.acc_y = req_thread->acc_y;
    complex_obj.acc_z = req_thread->acc_z;
    complex_obj.pitch = 180 *
atan(complex_obj.acc_x/sqrt(complex_obj.acc_y*complex_obj.acc_y +
complex_obj.acc_z*complex_obj.acc_z))/PI;
    complex_obj.roll = 180 *
atan(complex_obj.acc_y/sqrt(complex_obj.acc_x*complex_obj.acc_x +
complex_obj.acc_z*complex_obj.acc_z))/PI;
    complex_obj.yaw = 180 * atan
(complex_obj.acc_z/sqrt(complex_obj.acc_x*complex_obj.acc_x +
complex_obj.acc_z*complex_obj.acc_z))/PI;
    clock_gettime(CLOCK_REALTIME, &complex_obj.timestamp);

    pthread_mutex_unlock(&resource_lock);
}
```

```
void read_data(void *requesting_thread)
{
    obj_t *req_thread = (obj_t*) requesting_thread;
    pthread_mutex_lock(&resource_lock);

    req_thread->acc_x = complex_obj.acc_x;
    req_thread->acc_y = complex_obj.acc_y;
    req_thread->acc_z = complex_obj.acc_z;
    req_thread->roll = complex_obj.roll;
    req_thread->pitch = complex_obj.pitch;
    req_thread->yaw = complex_obj.yaw;
    req_thread->timestamp.tv_sec = complex_obj.timestamp.tv_sec;
    req_thread->timestamp.tv_nsec = complex_obj.timestamp.tv_nsec;

    printf("Acc_x = %lf\n",req_thread->acc_x);
    printf("Acc_y = %lf\n",req_thread->acc_y);
    printf("Acc_z = %lf\n",req_thread->acc_z);
    printf("Roll = %lf\n",req_thread->roll);
    printf("Pitch = %lf\n",req_thread->pitch);
    printf("Yaw = %lf\n",req_thread->yaw);
    printf("Timestamp: Sec = %lu, Nano Sec = %lu\n",req_thread-
>timestamp.tv_sec,req_thread->timestamp.tv_nsec);

    pthread_mutex_unlock(&resource_lock);
}

void task(void *requesting_thread)
{
    obj_t *req_thread = (obj_t*) requesting_thread;

    if(req_thread->id == 1)
    {
        srand(time(0));
        req_thread->acc_x = (double)((rand() % 256) + 12.567);
        req_thread->acc_y = (double)((rand() % 125) + 5.46);
        req_thread->acc_z = (double)((rand() % 50) + 45.67);
        calculate_data(req_thread);
        printf("Thread %ld - Finished Calculating Data\n",syscall(__NR_gettid));
    }
    else
    {
        printf("Thread %ld - Started Reading Data\n",syscall(__NR_gettid));
        read_data(req_thread);
        printf("Thread %ld - Finished Reading Data\n",syscall(__NR_gettid));
    }
}

int main()
{
    pthread_t thread1, thread2;
    int thread_status = 0;
    obj_t *obj1 = NULL;
    obj_t *obj2 = NULL;
```

```
obj1 = (obj_t*)malloc(sizeof(obj_t));
if(obj1 == NULL)
{
    printf("\nMalloc for Object-1 failed. Exiting\n");
    return 0;
}
obj1->id = 1;

obj2 = (obj_t*)malloc(sizeof(obj_t));
if(obj2 == NULL)
{
    printf("\nMalloc for Object-2 failed. Exiting\n");
    return 0;
}
obj2->id = 2;

if(pthread_mutex_init(&resource_lock, NULL) != 0)
{
    printf("\nMutex initialization failed. Exiting\n");
    return 0;
}

thread_status = pthread_create(&thread1, NULL, (void *)task, (void*) obj1);
if(thread_status != 0)
{
    printf("\nChild thread - 1 cannot be created\n");
    return 0;
}

thread_status = pthread_create(&thread2, NULL, (void *)task, (void*) obj2);
if(thread_status != 0)
{
    printf("\nChild thread - 2 cannot be created\n");
    return 0;
}

pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
pthread_mutex_destroy(&resource_lock);
return 0;}
```

### Output :

```
Thread 2768 - Finished Calculating Data
Acc_x = 120.567000
Acc_y = 119.460000
Acc_z = 76.670000
Roll = 39.918771
Pitch = 40.364363
Yaw = 28.232506
Timestamp: Sec = 1552076481, Nano Sec = 408233552
Thread 2769 - Finished Reading Data
```

### **Problem-3**

#### **Deadlock.c code description:**

In this code 2 threads are being executed. Thread 1 initially grabs resource A and locks the mutex on resource A. and thread 2 grabs the resource B and locks the mutex on resource B. now thread 1 tries to access the resource B , but it has been locked by thread 2 , similarly thread 2 tries to access resource A which has been locked by thread 1. Thus the system enters into a circular wait state where it is waiting for the resources that have been locked by other threads and hence deadlock occurs in the system as observed in the output below :

```
ubuntu@tegra-ubuntu:~/rtes/example-sync$ ./deadlock
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
rsrcACnt=0, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 grabbing resources
THREAD 1 got B, trying for A
THREAD 1 got A, trying for B
```

With safe:

```
ubuntu@tegra-ubuntu:~/rtes/example-sync$ ./deadlock 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: -1226779552 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: -1226779552 done
All done
ubuntu@tegra-ubuntu:~/rtes/example-sync$
```

With race:

```

ubuntu@tegra-ubuntu:~/rtes/example-sync$ ./deadlock race
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
rsrcACnt=0, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: -1226673056 done
Thread 2: -1235061664 done
All done
ubuntu@tegra-ubuntu:~/rtes/example-sync$

```

### **Root cause of deadlock:**

when there are 2 threads and the thread one acquires a resource and is waiting on a resource that is acquired by the 2<sup>nd</sup> thread, similarly, thread 2 acquires a resource and is waiting on a resource acquired by the 1<sup>st</sup> thread. Both the threads enter in a circular wait state and cannot come out if unless they get the resources. Hence the system stops and deadlock occurs in the system. There are ways to fix a deadlock by using a timedlock which holds the resource until a timeout occurs. Another way is to use a random backoff strategy a random time is specified after which the thread releases a resource and the other thread can acquire both the resources.

There is also a provision in the code so that the code can be executed by using 2 arguments 'safe' and 'race'.

When the safe argument is used the threads do not run simultaneously since a sleep time is added at the beginning of the thread. Hence thread 1 runs first completes its execution then it is joined to main, then thread 2 is created, it runs and completes its execution and then it is joined. hence no deadlock occurs in the system as we can observe in the output below:

When the race argument is used then the threads do not sleep and run immediately one after the other, here no deadlock is seen since the threads do not wait on any resource. We can see in the below image that no deadlock is observed, but also there is no message that says that each thread has acquired both the resources

### **Deadlock\_timeout.c code description:**

In this code a timeout strategy is used to avoid deadlock. A timeout is set for both Resource A and resource B. and pthread\_mutex\_timedlock is used. 1<sup>st</sup> the thread one is created and it spawns and acquires resource A, i.e locks the mutex on resource A. now if the unsafe test is run i.e 'safe' argument is not passed then this thread tries to immediately acquire resource B. now a timeout is set for resource B. and the thread waits until the timer expires, the macro ETIMEDOUT is used to check for the timeout error, if this returns an error then the mutex on resource A is unlocked. resource A is then free and can be accessed by thread 2. The same occurs with when thread 1 tries to acquire resource B. hence when a thread goes into timeout another thread gets a resource



and is able to complete its execution and hence deadlock is avoided. The output of this code is attached below.

```
[3] ~$ gcc -o deadlock -I../rtos/rtos
ubuntu@tegra-ubuntu:~/rtes/example-sync$ ./deadlock_timeout
Will set up unsafe deadlock scenario
Creating thread 1
Creating thread 2
will try to join both CS threads unless they deadlock
Thread 2 started
THREAD 2 grabbing resource B @ 1552077436 sec and 361700846 nsec
Thread 2 GOT B
rsrcACnt=0, rsrcBCnt=1
Thread 1 started
THREAD 1 grabbing resource A @ 1552077436 sec and 362567096 nsec
Thread 1 GOT A
rsrcACnt=1, rsrcBCnt=1
THREAD 2 got B, trying for A @ 1552077437 sec and 362502160 nsec
THREAD 1 got A, trying for B @ 1552077437 sec and 363254577 nsec
Thread 2 TIMEOUT ERROR
Thread 1 GOT B @ 1552077439 sec and 363133009 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
THREAD 1 got A and B
THREAD 1 done
Thread 1 joined to main
Thread 2 joined to main
All done
ubuntu@tegra-ubuntu:~/rtes/example-sync$
```

### **Deadlock solved.c code description:**

Another method called the random backoff strategy is used to solve the problem of deadlock. Here a random backoff time is set and the execution of the thread is delayed for that amount of time. The random time should be more than the time needed for the 1<sup>st</sup> thread to acquire both resources and complete the execution. After this thread 2 is spawned, it acquires both resources and completes its execution. Thus we can see in the below image that there is no deadlock involved.

```
cc -O -g -D_LINUX -o deadlock deadlock.c -pthread -lrt
ubuntu@tegra-ubuntu:~/rtes/example-sync$ ./deadlock
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
rsrcACnt=0, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: -1226087328 done
THREAD 2 grabbing resources
THREAD 1 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
Thread 2: -1234475936 done
All done
ubuntu@tegra-ubuntu:~/rtes/example-sync$
```

**Pthread3.c code description:**

In this code, there are services and they are allotted low, medium and high priorities. The service with low and high priority i.e. service 1 and 3 share a resource. The interference time is provided as an argument during run time, here an interference time of one second is provided. The low priority service first runs and locks the resource and then the high priority service has to be executed with also needs the same resource and a medium priority service occurs which attempts to preempt the low priority services. And the high priority service preempts the medium priority service. But since the resource is acquired by the lower priority service, the higher priority service doesn't run until the lower priority service is completed. From the below screenshot we can see that the lower priority service is completed at the end.

```
ubuntu@tegra-ubuntu:~/rtes/example-sync$ sudo ./pthread3
[sudo] password for ubuntu:
Usage: pthread interfere-seconds
ubuntu@tegra-ubuntu:~/rtes/example-sync$ sudo ./pthread3 1
interference time = 1 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
Creating thread 3
Low prio 3 thread spawned at 1552078567 sec, 519491 nsec
Start services thread spawned
will join service threads
Creating thread 2
Middle prio 2 thread spawned at 1552078568 sec, 520491 nsec
Creating thread 1, CScnt=1
High prio 1 thread spawned at 1552078568 sec, 520941 nsec
**** 2 idle stopping at 1552078568 sec, 521300 nsec
**** 3 idle stopping at 1552078569 sec, 519892 nsec
LOW PRIO done
MID PRIO done
**** 1 idle stopping at 1552078571 sec, 520124 nsec
HIGH PRIO done
START SERVICE done
All done
ubuntu@tegra-ubuntu:~/rtes/example-sync$
```

**Root cause of Priority Inversion:**

Priority inversion occurs when there are three threads with priority low, medium and high and the mutex i.e. shared resources for a particular critical section are between the higher priority and the lower priority services. when the critical section for the lower priority service is running and is holding the mutex on the shared resource and a medium priority thread occurs and tries to preempt it and the higher priority thread occurs and tries to preempt the medium thread, but there will be blocking of resource since the lower priority thread has the resource and it completes the execution first. Then a number of medium priority services can occur which will delay the execution of the higher priority service for an unbounded amount of time. This is the root cause for priority inversion. the priority inversion problem can be solved by priority inheritance and priority ceiling protocol.

**Pthread3ok.c code description:**

In this code the threads are created and executed in the order of their priorities, i.e the higher priority thread runs first, then the medium priority thread and then the lower priority thread runs. Thus since the threads run in this order there is no resource blocking and hence no priority inversion takes place. We can see that in the output below:

```
ubuntu@tegra-ubuntu:~/rtes/example-sync$ sudo ./pthread3ok 1
interference time = 1 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
Creating thread 1
High prio 1 thread spawned at 1552078637 sec, 938609 nsec
Creating thread 2
Middle prio 2 thread spawned at 1552078637 sec, 938660 nsec
Creating thread 3
Low prio 3 thread spawned at 1552078637 sec, 938705 nsec
**** 1 idle stopping at 1552078637 sec, 938734 nsec
**** 2 idle stopping at 1552078637 sec, 939014 nsec
**** 3 idle stopping at 1552078637 sec, 939063 nsec
LOW PRIO done
MID PRIO done
HIGH PRIO done
Start services thread spawned
will join service threads
START SERVICE done
All done
ubuntu@tegra-ubuntu:~/rtes/example-sync$
```

**For unbounded priority inversion, is there a real fix in linux, if not – why not ?**

In linux, there is no real fix for unbounded priority inversion without the RT\_PREEMPT\_PATCH. This patch is a part of linux kernel and supports priority inheritance.

**RT PREEMPT PATCH description & whether linux can be made real-time safe:**

The RT\_PREEMPT\_PATCH patch allows the kernel to be preempted, with some exceptions of small regions of code ("raw\_spinlock critical regions"). This is done by replacing most kernel spinlocks with mutexes that support priority inheritance. The main purpose of the RT\_PREEMPT\_PATCH is to minimize the amount of kernel code that is non preemptible. In particular, critical sections, interrupt handlers, and interrupt-disable code sequences are normally preemptible. This patch gives the ability to do priority inheritance. This can be done by setting priorities to system calls, interrupts etc. Since this patch makes kernel preemptible we need to handle the CPU variables carefully. Since the kernel is made preemptible, the higher priority

tasks that are ready will be executed immediately. Hence the concept of priority inheritance can be implemented to avoid priority inversion. Therefore when a low priority task is preempted by a high priority task then the priority of the higher priority task is inherited by the lower priority task. Thus a task having medium priority cannot interfere for an unbounded time. The critical section of the priority inherited task is executed immediately, after this is completed then the original priority is regained and thus the problem of priority inversion is solved by using this patch in linux and thus linux can be made real-time safe.

**Does it make sense to switch to an RTOS and not use Linux at all for both HRT and SRT services?**

In linux, even though the RT\_PREEMPT\_PATCH solves the problem of priority inversion, there is still a possibility that priority inversion occurs, this is due to the complexity of the Linux OS. Also, linux is not an OS designed for Hard real time systems. Hence, it makes sense to use an RTOS for both HRT and SRT services.

**Problem-4**

Following are the adapted versions from the “heap\_mq.c” and “posix\_mq.c” named as “native\_heap.c” and “native\_posix.c” respectively. Both codes have also been uploaded to the canvas with a Makefile.

**Code “native\_heap.c”:**

```
#include<fcntl.h>
#include<sys/stat.h>
#include<mqueue.h>
#include<stdio.h>
#include<stdlib.h>
#include<stdint.h>
#include<string.h>
#include<unistd.h>
#include<errno.h>
#include<sys/time.h>
#include<pthread.h>
#include<unistd.h>
#include<sys/syscall.h>
#include<signal.h>

#define Q_NAME          ("/my_queue1")
#define Q_SIZE          (8)

mqd_t msgqueue_FD;
struct mq_attr msgqueue_FD_attr;

static char imagebuff[4096];
static int sid, rid;

void sender(void)
{
    int value;

    char buffer[sizeof(void *)+sizeof(int)];
    void *buffptr;
    int id = 999;

    msgqueue_FD = mq_open(Q_NAME, O_CREAT | O_RDWR, 0666, &msgqueue_FD_attr);

    if(msgqueue_FD == (mqd_t)-1)
    {
        perror("Error in opening Message Queue");
        exit(1);
    }
}
```

```
while(1)
{
    buffptr = (void *)malloc(sizeof(imagebuff));
    strcpy(buffptr, imagebuff);
    printf("Message to send = %s\n", (char *)buffptr);

    printf("Sending %ld bytes\n", sizeof(buffptr));

    memcpy(buffer, &buffptr, sizeof(void *));
    memcpy(&(buffer[sizeof(void *)]), (void *)&id, sizeof(int));

    value = mq_send(msgqueue_FD, (char*)buffer, (size_t)(sizeof(void
*)+sizeof(int)),30);
    if(value == -1)
    {
        perror("Error in sending message");
    }
    else
    {
        printf("Send: message ptr 0x%p successfully sent\n", buffptr);
    }

    sleep(3);
}

void receiver(void)
{
    int value;

    char buffer[sizeof(void *)+sizeof(int)];
    void *buffptr;
    int count = 0;
    int id;
    int prio;

    msgqueue_FD = mq_open(Q_NAME, O_CREAT | O_RDWR, 0666, &msgqueue_FD_attr);
    if(msgqueue_FD == -1)
    {
        perror("Error in opennig message queue");
        exit(1);
    }

    while(1)
    {
        printf("Reading %ld bytes\n", sizeof(void *));

        value = mq_receive(msgqueue_FD, buffer, (size_t)(sizeof(void
*)+sizeof(int)),&prio);
        if(value == -1)
        {
            perror("Error in receiving message");
        }
        else
        {

```

```
        memcpy(&buffptr, buffer, sizeof(void *));
        memcpy((void *)&id, &(buffer[sizeof(void *)]), sizeof(int));
        printf("Receive: ptr msg %p, received with priority = %d, length = %d, id
= %d\n", buffptr,prio,value,id);

        printf("Contents of ptr = %s\n", (char *)buffptr);

        free(buffptr);

        printf("Heap space memory freed\n");
    }

}

void _handler_kill(int signal)
{
    printf("Killed by Ctrl-C\n");
    mq_close(msgqueue_FD);
    mq_unlink(Q_NAME);
    exit(1);
}

int main()
{
    pthread_t thread1, thread2;
    int thread_status = 0;

    pthread_attr_t child1_attr, child2_attr;
    struct sched_param child1_param, child2_param, main_param;

    sched_getparam(getpid(), &main_param);
    int max_prio = sched_get_priority_max(SCHED_FIFO);
    main_param.sched_priority = max_prio;
    sched_setscheduler(getpid(), SCHED_FIFO, &main_param);

    pthread_attr_init(&child1_attr);
    pthread_attr_init(&child2_attr);

    pthread_attr_setschedpolicy(&child1_attr, SCHED_FIFO);
    pthread_attr_setschedpolicy(&child2_attr, SCHED_FIFO);

    child1_param.sched_priority = max_prio-1;
    child2_param.sched_priority = max_prio-2;

    pthread_attr_setschedparam(&child1_attr, &child1_param);
    pthread_attr_setschedparam(&child2_attr, &child2_param);

    struct sigaction kill_action;
    memset (&kill_action, 0, sizeof (kill_action));
    kill_action.sa_handler = _handler_kill;
    sigaction (SIGINT, &kill_action, NULL);

    msgqueue_FD_attr.mq_maxmsg = Q_SIZE;
    msgqueue_FD_attr.mq_msgsize = sizeof(void *)+sizeof(int);
```

```
int i, j;
char pixel = 'A';

for(i = 0 ; i < 4096 ; i += 64)
{
    pixel = 'A';
    for(j = i ; j < i + 64 ; j++)
    {
        imagebuff[j] = (char)pixel++;
    }
    imagebuff[j-1] = '\n';
}
imagebuff[4095] = '\0';
imagebuff[63] = '\0';

printf("buffer =%s\n", imagebuff);

thread_status = pthread_create(&thread1,&child1_attr,(void *)receiver, NULL);
if(thread_status != 0)
{
    printf("\nChild thread - 1 cannot be created\n");
    return 0;
}

thread_status = pthread_create(&thread2,&child2_attr,(void *)sender, NULL);
if(thread_status != 0)
{
    printf("\nChild thread - 2 cannot be created\n");
    return 0;
}

pthread_join(thread1, NULL);
pthread_join(thread2, NULL);

return 0;
}
```



**Output:**

```

ridhi@ridhi:~/RTES/Problem4$ ./native_heap
buffer =ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Reading 8 bytes
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
Send: message ptr 0x0x7f4214000b20 successfully sent
Receive: ptr msg 0x7f4214000b20, received with priority = 30, length = 12, id = 999
Contents of ptr = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Heap space memory freed
Reading 8 bytes
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
Send: message ptr 0x0x7f4214000b20 successfully sent
Receive: ptr msg 0x7f4214000b20, received with priority = 30, length = 12, id = 999
Contents of ptr = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Heap space memory freed
Reading 8 bytes
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
Send: message ptr 0x0x7f4214000b20 successfully sent
Receive: ptr msg 0x7f4214000b20, received with priority = 30, length = 12, id = 999
Contents of ptr = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Heap space memory freed
Reading 8 bytes
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
Send: message ptr 0x0x7f4214000b20 successfully sent
Receive: ptr msg 0x7f4214000b20, received with priority = 30, length = 12, id = 999
Contents of ptr = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Heap space memory freed
Reading 8 bytes
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
Send: message ptr 0x0x7f4214000b20 successfully sent
Receive: ptr msg 0x7f4214000b20, received with priority = 30, length = 12, id = 999
Contents of ptr = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Heap space memory freed
Reading 8 bytes
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

```

**Code “native\_posix.c”:**

```

#include<fcntl.h>
#include<sys/stat.h>
#include<mqueue.h>
#include<stdio.h>
#include<stdlib.h>
#include<stdint.h>
#include<string.h>
#include<unistd.h>
#include<errno.h>
#include<sys/time.h>
#include<pthread.h>

```

```
#include<unistd.h>
#include <sched.h>
#include<sys/syscall.h>

#define Q_NAME          ("/my_queue1")
#define Q_SIZE          (8)

typedef struct Message
{
    char string[200];
    int length;
}mesg_t;

mqd_t msgqueue_FD;
struct mq_attr msgqueue_FD_attr;

void sender(void)
{
    char str[200];
    int value;

    mesg_t message;
    mesg_t *msgptr;

    msgqueue_FD = mq_open(Q_NAME, O_CREAT | O_RDWR, 0666, &msgqueue_FD_attr);

    if(msgqueue_FD == (mqd_t)-1)
    {
        perror("Error in opening Message Queue");
        exit(1);
    }

    sprintf(str, "%s %ld","this is a test, and only a test, in the event of a real
emergency, you would be instructed ... from Task ",syscall(__NR_gettid));
    strcpy(message.string, str);
    message.length = strlen(message.string);
    msgptr = &message;

    value = mq_send(msgqueue_FD, (char*)msgptr, sizeof(mesg_t),30);
    if(value == -1)
    {
        perror("Error in sending message");
        exit(1);
    }
}

void receiver(void)
{
    int value;
    char str[100];
    int prio;

    mesg_t message;
    mesg_t *msgptr;
```

```
msgqueue_FD = mq_open(Q_NAME, O_CREAT | O_RDWR, 0666, &msgqueue_FD_attr);
if(msgqueue_FD == -1)
{
    perror("Error in opennig message queue");
    exit(1);
}

msgptry = &message;

value = mq_receive(msgqueue_FD, (char *)msgptry, sizeof(mesg_t), &prio);
if(value == -1)
{
    perror("Error in receiving message");
    exit(1);
}

sprintf(str, "%s %ld\n", "Received by Task", syscall(__NR_gettid));
printf("%s%s, lenght = %d, Priority = %d\n", str, msgptry->string, msgptry->length,
prio);
}

int main()
{
    pthread_t thread1, thread2;
    int thread_status = 0;

    pthread_attr_t child1_attr, child2_attr;
    struct sched_param child1_param, child2_param, main_param;

    sched_getparam(getpid(), &main_param);
    int max_prio = sched_get_priority_max(SCHED_FIFO);
    main_param.sched_priority = max_prio;
    sched_setscheduler(getpid(), SCHED_FIFO, &main_param);

    pthread_attr_init(&child1_attr);
    pthread_attr_init(&child2_attr);

    pthread_attr_setschedpolicy(&child1_attr, SCHED_FIFO);
    pthread_attr_setschedpolicy(&child2_attr, SCHED_FIFO);

    child1_param.sched_priority = max_prio-1;
    child2_param.sched_priority = max_prio-2;

    pthread_attr_setschedparam(&child1_attr, &child1_param);
    pthread_attr_setschedparam(&child2_attr, &child2_param);

    msgqueue_FD_attr.mq_maxmsg = Q_SIZE;
    msgqueue_FD_attr.mq_msgsize = sizeof(mesg_t);

    thread_status = pthread_create(&thread1, &child1_attr, (void *)receiver, NULL);
    if(thread_status != 0)
    {
        printf("\nChild thread - 1 cannot be created\n");
        return 0;
    }
}
```

```

    thread_status = pthread_create(&thread2,&child2_attr,(void *)sender, NULL);
    if(thread_status != 0)
    {
        printf("\nChild thread - 2 cannot be created\n");
        return 0;
    }

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    mq_close(msgqueue_FD);
    mq_unlink(Q_NAME);

    return 0;
}

```

### Output :

```

es  Terminal  Fri 13:48
souvik@souvik-vb: Problem4
File Edit View Search Terminal Help
souvik@souvik-vb:~$ cd RTE/Exercise3/Problem4
souvik@souvik-vb:~/RTE/Exercise3/Problem4$ make
gcc -pthread native_posix.c -o native_posix -lrt
gcc -pthread native_heap.c -o native_heap -lrt
souvik@souvik-vb:~/RTE/Exercise3/Problem4$ ./native_posix
Received by Task 2177
this is a test, and only a test, in the event of a real emergency, you would be instructed ... from Task 2178, lenght = 110, Priority = 30
souvik@souvik-vb:~/RTE/Exercise3/Problem4$ ./native_posix
Received by Task 2180
this is a test, and only a test, in the event of a real emergency, you would be instructed ... from Task 2181, lenght = 110, Priority = 30
souvik@souvik-vb:~/RTE/Exercise3/Problem4$

```

### **Description of how the message queues address the problem of shared global memory:**

Both programs use message queues to send unidirectional data from the sender to the receiver. The sender and the receiver are initiated by two different child threads, with the priority of the thread subsequent to the receiver task being higher than the thread corresponding to the sender. The receiver is assigned a higher priority in both the programs as we don't want it to miss any messages from the sender. Reception is a blocking protocol for Message Queues, meaning that the receiver thread will wait on the read instruction until a message is intercepted from the sender. It is therefore imperative that the receiver thread must run first to ensure that no message from the sender is lost in transmission.

The objective of the pthread adaptation of "posix\_mq.c" i.e. "native\_posix.c" is to send a string from the sender to the receiver and then displaying it on the receiver thread. In order to achieve a robust solution, we decided to send a structure object instead, giving us more versatility, if we decide to modify the program to our needs in the future. The structure object comprises of two members, a string and its length. We send a string literal, its length and the thread identity of the sender thread from the sender side. On the receiver thread, we display the subsequent information along with the message queue priority assigned at the sender side and the receiver thread id.

The objective of the pthread adaptation of the “heap\_mq.c” i.e. “native\_heap.c” is to build a string of ASCII character from ‘A’ to ‘~’, i.e. starting from character ‘A’ and traversing 64 other characters after that. This string is sent from the sending thread over to the receiver repeatedly using an infinite while loop with a delay of three seconds between every transmission. The receiver receives the message and displays the subsequent message along with its length and the message queue priority set by the sender. The delay is placed at the sender side to ensure that there is no loss in the transmission and that the receiver can successfully receive all sent messages. Since the communication process lasts indefinitely, interrupting the process to close it using Ctrl-C doesn’t close and unlink the message queue. We have used the SIGINT signal to kill the process and close the message queue in an efficient way using Ctrl-C, to avert any inadvertent exceptions.

In both the programs, receiver and sender threads have been scheduled as FIFO with the receiver thread having a higher priority than the sender thread.

Message Queues could be used as a means of sending data from one thread/process to another, thereby making the use of global and shared variables redundant. In other words, it could reduce the use of critical resources, which otherwise would require mutex. In the programs above, our receiver thread had a higher priority than the sender. It is known that the receiving thread blocks until data is received from the sender. If we go by this example, although the higher priority task is dependent on the lower priority task, it could run until it is expecting data from the sender to complete its task. At which point it blocks on read while the sender is getting ready to send the required data. But if a medium priority task comes and preempts the lower priority task, the higher priority receiver thread, would be blocked until the execution of the medium priority task is finished. Hence, we conclude, although message queues are efficient in reducing dependency on critical resources, they cannot be used to avert unbounded priority inversion.

**Problem 5:**

Watchdog timers are used by the Linux Watchdog Daemon as a device which has a hardware timer which will reset the system on completion of its timer. The Daemon can be used as a stand-alone system monitoring system which will make use of many tests to make sure the system is working as predicted.

The watchdog daemon has about 4 settings for watchdog timer devices. The first two settings give the API point of the timer device and the watchdog time-out. In our case, the daemon opens /dev/watchdog and keeps writing to the location before every 60 seconds, so that the timer does not run out and cause the system from resetting. The next setting defines the polling interval of the system whose value is by default is 1 second. The interval defines the time the system sleeps between the loops defined for checking the proper functionality of the system. The selection of the timeout duration and polling interval is very critical to the operation of the system. The timeout duration should be calculated by considering the worst-case scenarios and to avoid having deadlocks. The polling interval can be computed so that they have a low power utilization and do not reduce the chances of detecting problems at the earliest.

Also, there is logging of errors done by the watchdog daemon, it by default prints out the log messages to syslog. Storing the syslog file on cloud or some central device for storing them for analysis in case of any failures causing the system to lose the data.

In systems where routines can call back and forth between them, any situation where two or more threads can be stuck waiting for the locks held by one another. For example, if we consider two threads which utilizes two locks individually and then wait for the lock held by the other thread to get available, this situation is called as a Deadlock. Once the threads have deadlocked, they won't execute further code, other than such code as may be involved in waiting on the two locks.

As an example, consider a program that has two threads, T1 and T2, and two locks, L1 and L2. A deadlock occurs in the following scenario:

Time	Thread T1	Thread T2
0 ms	in FuncB()	in FuncA()
10 ms	acquires L1	acquires L2
15 ms	calls FuncA()	calls FuncB()
20 ms	waits for L2	waits for L1
30 ms		
60 ms	(a hang makes itself obvious)	
120 ms		
...		

In case of a deadlock, the system will have no resources to execute the tasks. In such cases where all resources are blocked, the system processes will not be able to feed the watchdog timer and thus resulting in reset of the whole system. As the system resets, the resources become free once again and hence the system is out of deadlock. Thus, watchdog timers in Linux are very useful when there is a deadlock condition in a system or program.

### **Timeout:**

To explore the timeout function, the code from 2 was modified. The `timedlock` was used to lock the mutex, a ten second time was set. And it was checked that after every 10 seconds if new data was available. The new data not available is printed after every 10 seconds.

In the below output we can see that first, the data has been calculated i.e updated, then we check for the new data again, after 10 seconds no new data available is printed. We can the time difference between the 2 messages is 10 seconds.

The following part in the function was modified :

In the read function a timeout was set using the `timespec` variable `times`. The timer was set for 10 mins, then a `pthread_mutex_timedlock()` was used and the timer argument was passed in this.

The `pthread_mutex_timedlock()` function locks the mutex object referenced by `mutex`. If the mutex is already locked, the calling thread blocks until the mutex becomes available. If the mutex cannot be locked without waiting for another thread to unlock the mutex, this wait will be terminated when the specified timeout expires. If successful, the `pthread_mutex_timedlock()` function returns zero, otherwise an error number is returned to indicate the error. If this function returns 0 then the accelerometer read is successful and the accelerometer values are printed. If there is an error and timeout occurs we print that no new data is available.

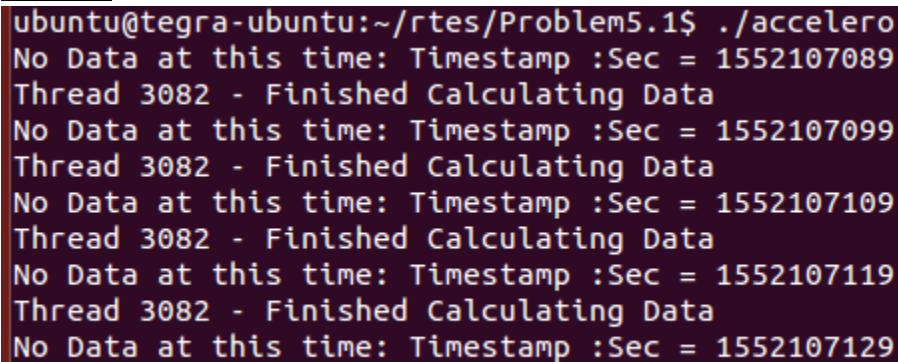
```
void read_data(void *requesting_thread)
{
    struct timespec times;
    clock_gettime(CLOCK_REALTIME, &times);
    times.tv_sec += 10;
    obj_t *req_thread = (obj_t*) requesting_thread;
    int a = pthread_mutex_timedlock(&resource_lock, &times);

    if(a == 0)
    {
        req_thread->acc_x = complex_obj.acc_x;
        req_thread->acc_y = complex_obj.acc_y;
        req_thread->acc_z = complex_obj.acc_z;
        req_thread->roll = complex_obj.roll;
        req_thread->pitch = complex_obj.pitch;
        req_thread->yaw = complex_obj.yaw;
        req_thread->timestamp.tv_sec = complex_obj.timestamp.tv_sec;
        req_thread->timestamp.tv_nsec = complex_obj.timestamp.tv_nsec;

        printf("Acc_x = %lf\n", req_thread->acc_x);
        printf("Acc_y = %lf\n", req_thread->acc_y);
        printf("Acc_z = %lf\n", req_thread->acc_z);
        printf("Roll = %lf\n", req_thread->roll);
        printf("Pitch = %lf\n", req_thread->pitch);
        printf("Yaw = %lf\n", req_thread->yaw);
        printf("Timestamp: Sec = %lu, Nano Sec = %lu\n", req_thread->timestamp.tv_sec, req_thread->timestamp.tv_nsec);

        pthread_mutex_unlock(&resource_lock);
    }
    else
    {
        if(clock_gettime(CLOCK_REALTIME, &(times)) == -1)
            perror("clock_gettime failed");
        printf("No Data at this time: Timestamp :Sec = %ld", times.tv_sec);
    }
}
```

### Output:



```
ubuntu@tegra-ubuntu:~/rtes/Problem5.1$ ./accelero
No Data at this time: Timestamp :Sec = 1552107089
Thread 3082 - Finished Calculating Data
No Data at this time: Timestamp :Sec = 1552107099
Thread 3082 - Finished Calculating Data
No Data at this time: Timestamp :Sec = 1552107109
Thread 3082 - Finished Calculating Data
No Data at this time: Timestamp :Sec = 1552107119
Thread 3082 - Finished Calculating Data
No Data at this time: Timestamp :Sec = 1552107129
```



**References:**

- <http://www.drdobbs.com/parallel/deadlock-proof-your-code-part-1/225400066>
- <https://lwn.net/Articles/146861/>
- <https://lwn.net/Articles/178253/>
- <https://lwn.net/Articles/177111/>
- Real-Time Embedded Components and Systems Using Linux and RTOS by Sam siewert and John Pratt
- Linux man pages
- Priority Inheritance Protocols: An Approach to Real-Time Synchronization , by Rajkumar, sha, lechozky