University of Colorado Boulder

Harsimransingh Bindra, Nagarjuna Reddy

ECEN 5623 Real Time Embedded Systems

Exercise 3 3/8/2018

Question 1
Read Sha, Rajkumar, et al paper, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization" and summarize 3 main key points the paper makes. Read Dr. Siewert's summary paper on the topic as well. Finally, read the positions of Linux Torvalds as described by Jonathan Corbet and Ingo Molnar and Thomas Gleixner on this topic as well. Take a position on this topic yourself and write at least one well supported paragraph or more to defend your position based on what we have learned in class. Does the PI-futex (Futex, Futexes are Tricky) that is described by Ingo Molnar provide safe and accurate protection from un-bounded priority inversion as described in the paper? If not, what is different about it?

Solution:
Three key points which are discussed in the paper
1. The Priority Inversion Problem
2. Basic Priority Inheritence protocol
3. The Priority Ceiling Protocol

The Priority Inversion Problem
In a normal scenario a higher priority task preempts a lower priority task if a higher priority task is available in ready queue. But when we use primitive synchronisations for protecting the critical sections and protect shared memory from data corruption the scenario changes. Here If the lower priority task gains access to shared data first and locks it using a synchronisation primitive and now a higher priority tasks enter the ready queue and preempts the lower priority task, when it tries to access the shared data which is locked by the lower priority task, the higher priority task gets blocked by a lower priority task until the lower priority task completes its access to the shared data. This scenario can be simply called as blocking which is a form of priority inversion. Prolonged duration of blocking can cause the services to miss their deadlines. The above example can be dealt normally just by calculating the time for which the higher priority task is blocked and add it in its worst case execution time. The above case is an example of bounded priority inversion where the blocking time can be determined. But if there unbounded priority inversion then the WCET cannot be calculated and the chances of a higher priority job to miss its deadline is extremely high. For example, If there are three jobs J1, J2, J3 with priority H(High), M(Medium), L(Low) respectively. J1 and J3 have a common critical section and J2 is independent then an unbounded priority inversion can occur in this case. Suppose J3 gains access to the shared data and locks it, then J1 will be blocked by J3 till J3 has access to the shared data. Now while J3 is in the critical section and J2 is placed in the ready queue, J2 will preempt J3 as J2 has higher priority. Now J1 is blocked by J3 and J3 is preempted by J2. hence J1 now is blocked for a longer time than the normal blocking time calculated in the first scenario. The situation can worsen if J2 occurring frequency is high and it keeps on preempting J3 and never allow it to come out of its critical section. This scenario is called unbounded priority inversion which can be catastrophic in case of Hard real-time systems as high priority job is sure to miss its deadline. To overcome the priority inversion Problem there are two protocols discussed in the paper.

Basic Priority Inheritance Protocol

The main principle of basic priority inheritance protocol mentioned in the paper is, if a lower priority task blocks a higher priority task then the higher priority task lends it priority to the lower priority task till the time it is executing in the critical section. This will prevent any medium priority task from preempting the lower priority task and would prevent unbounded priority inversion. As soon as the lower priority task is done with execution of its critical section it gives up the higher priority which means that now the higher priority task can preempt the lower priority task and finish its execution. This protocol deals with the issue of priority inversion but cannot deal with some other issues mentioned in the paper. The issues with using basic priority inheritance protocol are as follows

- Deadlocks occurring due to nested critical sections
- Chained blocking

Deadline occurring due to nested critical sections

Suppose there are 4 services present namely J1,J2, J3 and J4 with priorities from highest to lowest respectively. There are 2 semaphores S1 and S2 shared between all the jobs. First J3 starts and captures the semaphore S1 and begins its execution. Now Job J2 starts and acquires semaphore S2 initially and now tries to acquire semaphore S1. Semaphore S1 is acquired by Job J3 so Job J2 gets blocked. Now when J3 tries to acquire semaphore S2 which is acquired by J2, J3 gets blocked. Now all the services are blocked as all the services require both the semaphores to complete its execution. This causes the system to halt indefinitely as no service can complete its execution and all services are in blocked state. This situation is not dealt with in the basic priority inheritance protocol

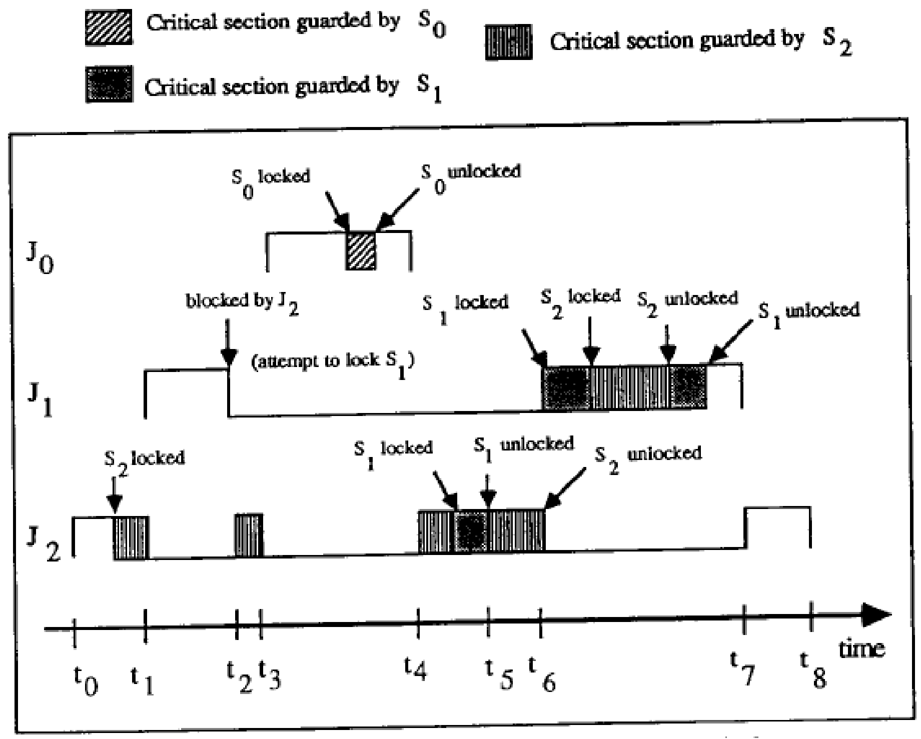
Chained Blocking

Suppose there are 4 services present namely J1,J2, and J3 with priorities from highest to lowest respectively. There are 2 semaphores S1 and S2 shared between (J1,J3) and (J1,J2)respectively. Now J3 starts first and acquires S1. Then J2 starts running and acquires semaphore S2. Now, according to the basic priority inheritance protocol J1 needs to wait for completion of 2 critical sections, one of J3 and one of J2. This is a case of chained blocking where a higher priority job has to wait for multiple critical sections before it acquires the semaphores. This can cause the higher priority job to miss its deadline.

These 2 problems present in basic priority inheritance protocol are dealt with in the priority ceiling protocol.

Priority Ceiling Protocol

The aim of this protocol is to avoid the deadlocks and chained blocking issues present in the basic priority inheritance protocol. Basic principle of this protocol is to ensure that when a task preempts the critical section of another task and tries to execute its own critical section, the priority at which the new task's critical section will execute should have higher priority than the inherited priorities of the preempted critical section. If this condition is not met then the new task will not be allowed to enter the critical section and the task whiched block this new task will inherit this new task's priority. This protocol is implemented by first calculating the ceiling priority for all the semaphores, which is equal to the highest priority task that uses that particular semaphore. Then a particular task is only allowed to enter the critical section if its priority is higher than all the priority ceilings of all the other semaphores which are locked by other tasks. The working of priority ceiling protocol can easily be understood by observing the following diagram



In the above diagram, J2 acquires semaphore S2 and then it is preempted by J1. J1 tries to lock S1 but fails to do so as its priority is not higher than the ceiling priority of semaphore S2. Hence J1 gets blocked and J2 resumes its execution. Then J0 preempts J2 and starts executing and is not blocked by other jobs as it only requires S0 for execution which is available. It completes its execution. After this J2 resumes its execution and completes executing its critical section. As soon as J2 releases both the semaphores J1 preempts J2 and acquires semaphore S1 and S2. It completes its execution and then J2 resumes again to finish finally.

Why the Linux position makes sense or not

According to me, Linus's explanation of not using PI in Kernel space seems logical because of the excess overhead generated due to multiple System calls made to the kernel, which hampers the performance of the system.Linus's clearly states in his article that
"Friends don't let friends use priority inheritance"
"Just don't do it. If you really need it, your system is broken anyway."
Linus strongly opposes to use Priority Inheritance, he mentions many other approaches to avoid priority inversions such as using lockless designs or carefully thought-out locking scenarios. He mentions that even though priority inheritance is simple it implement there are a lot of drawbacks associated with it such as Priority Inheritance tends to complicate and slow down the locking process of the code and it is not a complete remedy to priority inheritance problem.

When most of the developers avoided use of priority inheritance, Inglo, on the contrary posted a priority in inheriting futex implementation. He proposed a Lightweight Priority Inheritance Scheme in the userspace while avoiding the problems faced in other kernel space methods. User-space PI help achieving/improving determinism for user-space applications. In the best-case, it can help achieve determinism and well-bound latencies. Even in the worst-case, PI will improve the statistical distribution of locking related application delays. Three main reasons why it is called lightweight is mentioned in the article are[1]

- in the user-space fastpath a PI-enabled futex involves no kernel work (or any other PI complexity) at all. No registration, no extra kernel calls - just pure fast atomic ops in userspace.
- in the slowpath (in the lock-contention case), the system call and scheduling pattern is in fact better than that of normal futexes, due to the 'integrated' nature of FUTEX_LOCK_PI.
- the in-kernel PI implementation is streamlined around the mutex abstraction, with strict rules that keep the implementation relatively simple: only a single owner may own a lock (i.e. no read-write lock support), only the owner may unlock a lock, no recursive locking, etc.

Implementation

the userspace fastpath of PI-enabled pthread mutexes doesn't involve the kernel - they work in a very similar fashion to normal futex-based locks:a null value corresponds to unlocked, and a value==TID means that it is locked. Userspace uses atomic ops to lock/unlock these mutexes without entering the kernel.

There are two futex ops added in the patch to handle the slowpath

FUTEX_LOCK_PI

FUTEX_UNLOCK_PI

Finally after considering all the points according to us using Priority Inheritance futexes is a good option as it avoids involving kernel as much as possible using the fastpaths and overcomes the problems caused by the other previous synchronization primitives like Mutexes which are completely implemented in kernel space, while it tries to solve the problem of unbounded priority inheritance.

Reasoning on whether FUTEX fixes unbounded Priority Inversion

Futex is a good option to avoid priority inversion in user space. There are three main advantages.

The following three reasons makes PI Futex different from other primitive synchronisation techniques[1]

- As It is implemented in the user-space fastpath a PI-enabled futex involves no kernel work (or any other PI complexity) at all. There is no registration neither any extra kernel calls - just pure fast atomic ops in userspace.
- In the slowpath (in the lock-contention case), the system call and the scheduling pattern in fact performs better than that of normal futexes, due to the 'integrated' nature of FUTEX_LOCK_PI.
- the in-kernel PI implementation is streamlined around the mutex abstraction, with strict rules that keep the implementation relatively simple: only a single owner may own a lock (i.e. no read-write lock support), only the owner may unlock a lock, no recursive locking, etc.

A futex is simply defined as (short for "fast userspace_mutex") is a_kernal system call that is generally used to implement locking, or as a building block for higher-level locking abstractions such as_conditional variables, semaphores and_posix mutexes.[2] When a Futex-based lock is properly programmed it will not use system calls except when the lock is contended, most of the operations don't require arbitration between processes, this will not happen in most cases.The PI futex has the same basic principle as the basic priority inheritance protocol. If the PI futexes are used and if the priority of the task which is blocking a current higher priority task, than in this situation task which currently has access to the lock is boosted to the priority of the higher priority task. This boost is transitive as mentioned in the paper and if the process which is currently holding the futex is blocked on another futex, the boosted priority will be now be given to the second futex, and as soon as their critical section execution is finished everyone will get back their original priorities. Hence according to me they can be implemented to solve the unbounded priority inversion problem. But one thing should be kept in mind that it does not address problems such as chained blocking and deadlocks.[3]

Question2:
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).

Solution:
Description on re-entrancy and thread-safety:
Reentrancy:
A function is called reentrant if it can be interrupted in the middle of its execution and then safely be called again ("re-entered") before its previous invocations complete execution. The interruption could be caused by an internal action such as a jump or call, or by an external action such as an interrupt or signal. Once the reentered invocation completes, the previous invocations will resume correct execution. A reentrant function does not hold static data over successive calls, nor does it return a pointer to static data. All data is provided by the caller of the function. A reentrant function must not call non-reentrant functions.[6]

Thread Safety:
Thread safety is a computer programming concept applicable to multi-threaded code. Thread-safe code only manipulates shared data structures in a manner that ensures that all threads behave properly and fulfill their design specifications without unintended interaction. There are various strategies for making thread-safe data structures such as MUTEX, semaphore and semlock synchronizations.[7]

Description of methods to achieve reentrancy & thread safety, how to code them and its impact on real-time threads/tasks:
There are three basic methods to handle data in a_thread safe & reentrant way. And, we described each method below with the way to implement code for them and its impacts on real-time services.

The first method is, don’t use shared or global data in_the function. Instead use only local variables (stack). These functions are called ‘pure functions.’ To implement code in this method, we need to make sure we don’t use the static keyword and also make sure that no data is declared in the global scope. The goal here is to make sure that no state gets created that can be corrupted by a reentrant call to a function. As, we are not using any shared memory in this method, it won’t have much impact on real time threads and tasks. Different tasks would be able to run and preempt one another without an effect on the data consistency or data from other tasks.

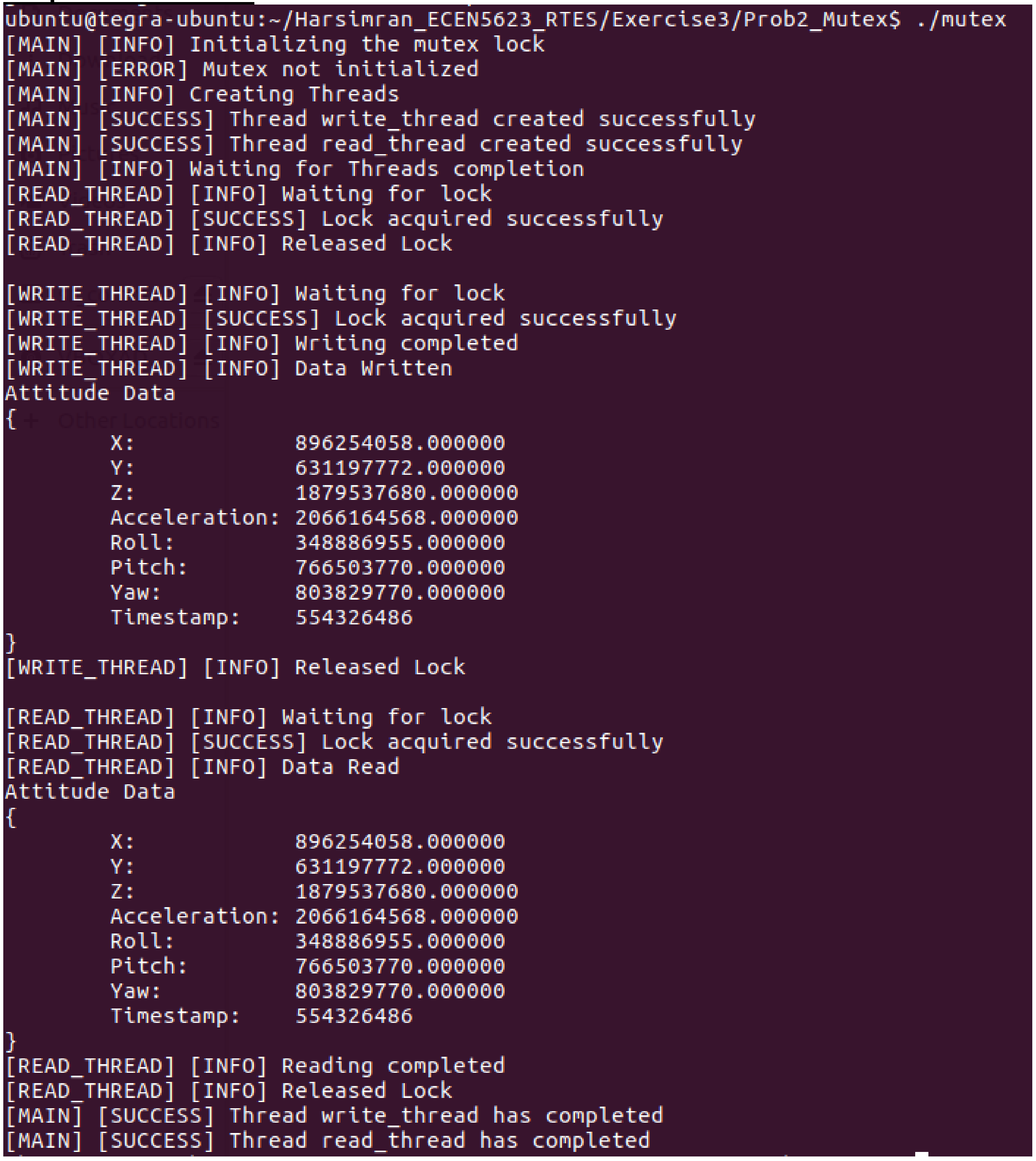
The second method is, use global data, but localizes it to a single thread or function. In this way, static or global data can be held without risking data corruption. To implement code in this method, we need to set up an indexed global data space based on thread. This requires us to think about how many threads we will need to handle beforehand, and create a data structure that is conducive to indexed global access. As, we are using thread indexed global data in this method, we can get into trouble if a task ever preempted itself. The functions in this case would have to be written in such a way that the data would not be corrupted even if the function is called in a re-entrant manner. Being said that, this shouldn't happen in a real time embedded system where tasks should finish before the next call to the task occurs.

The third method is, to use shared global data but to synchronize thread access using a mutex, semaphore, or data lock mechanisms. To implement code in this method, we need to create some sort of synchronization scheme that will protect critical sections of data during reentrant function calls. If we are using linux pthreads, this will be accomplished using the pthreads_mutex_t type with pthread_mutex_lock(), pthread_mutex_timedlock() and pthread_mutex_unlock() functions. An example of this idea will be shown later in this problem. This method needs a serious consideration in a real time embedded system. The shared global data can easily be corrupted when a low priority task is interrupted by a higher priority task that uses the same shared data. Real time systems need to have the ability to interrupt tasks in this manner. That means that a MUTEX or other data locking scheme is necessary to protect data despite task interruption. There are other considerations to this, however, such as a deadlock or priority inversion (described earlier). These need to be carefully considered when using the shared global data scheme, as you can easily create a situation in which no work is completed and deadlines are missed.

Thread safe code using MUTEX:

In our implementation, we have two threads running concurrently and accessing the shared data "attitude". One thread (write_thread) is writing data to it and the other thread (read_thread) is reading the data from it. And in this scenario, to provide protection from simultaneous access to the shared data "attitude" we implemented MUTEX lock synchronization mechanism. In this method, one thread acquires the mutex lock and access the shared data while the other thread waits for the lock until it is freed by the other thread. So, in this way only one thread can access the shared data at a time providing protection from data corruption due to simultaneous access.

Output screen shot:



```
ubuntu@tegra-ubuntu:~/Harsimran_ECEN5623_RTES/Exercise3/Prob2_Mutex$ ./mutex
[MAIN] [INFO] Initializing the mutex lock
[MAIN] [ERROR] Mutex not initialized
[MAIN] [INFO] Creating Threads
[MAIN] [SUCCESS] Thread write_thread created successfully
[MAIN] [SUCCESS] Thread read_thread created successfully
[MAIN] [INFO] Waiting for Threads completion
[READ_THREAD] [INFO] Waiting for lock
[READ_THREAD] [SUCCESS] Lock acquired successfully
[READ_THREAD] [INFO] Released Lock

[WRITE_THREAD] [INFO] Waiting for lock
[WRITE_THREAD] [SUCCESS] Lock acquired successfully
[WRITE_THREAD] [INFO] Writing completed
[WRITE_THREAD] [INFO] Data Written
Attitude Data
{
        X:              896254058.000000
        Y:              631197772.000000
        Z:              1879537680.000000
        Acceleration:   2066164568.000000
        Roll:           348886955.000000
        Pitch:          766503770.000000
        Yaw:            803829770.000000
        Timestamp:      554326486
}
[WRITE_THREAD] [INFO] Released Lock

[READ_THREAD] [INFO] Waiting for lock
[READ_THREAD] [SUCCESS] Lock acquired successfully
[READ_THREAD] [INFO] Data Read
Attitude Data
{
        X:              896254058.000000
        Y:              631197772.000000
        Z:              1879537680.000000
        Acceleration:   2066164568.000000
        Roll:           348886955.000000
        Pitch:          766503770.000000
        Yaw:            803829770.000000
        Timestamp:      554326486
}
[READ_THREAD] [INFO] Reading completed
[READ_THREAD] [INFO] Released Lock
[MAIN] [SUCCESS] Thread write_thread has completed
[MAIN] [SUCCESS] Thread read_thread has completed
```

Question 3

Download http://mercury.pr.erau.edu/~siewerts/cec450/code/example-sync/ and describe both the issues of deadlock and unbounded priority inversion and the root cause for both in the example code. Fix the deadlock so that it does not occur by using a random back-off scheme to resolve. For the unbounded inversion, is there a real fix in Linux – if not, why not? What about a patch for the Linux kernel? For example, Linux Kernel.org recommends the RT Preempt Patch, but would this really help? Read about the patch and describe why think it would or would not help with unbounded priority inversion. Based on inversion, does it make sense to simply switch to an RTOS and not use Linux at all for both HRT and SRT services?

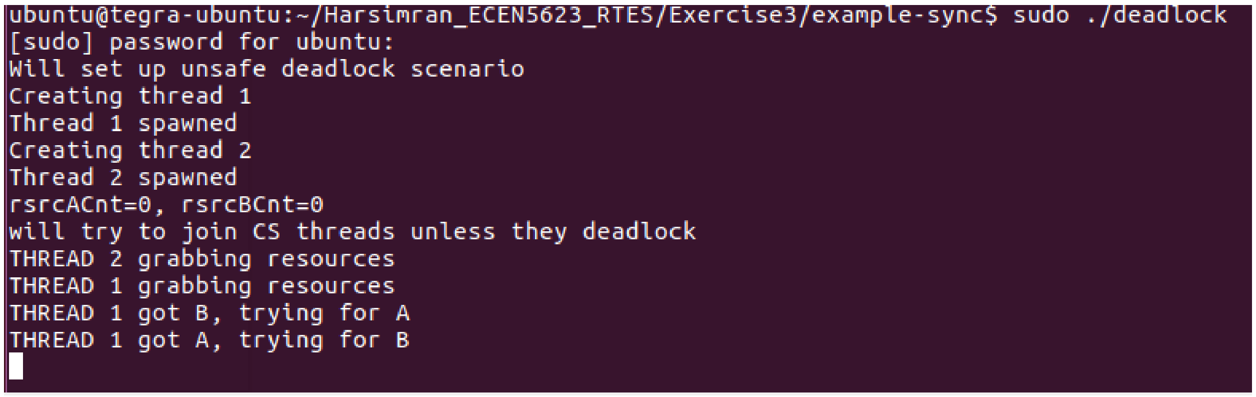
Solution:

<u>Demonstration and description of deadlock with threads</u>

A deadlock can be simply defined as situation in which two tasks sharing the same resources are blocking each other from accessing the resource, resulting in both tasks ceasing to function. This case can be observed in the first case of the given example.[4]

The output of the deadlock code which is execute in unsafe condition is provided below

Deadlock Screenshot



```
ubuntu@tegra-ubuntu:~/Harsimran_ECEN5623_RTES/Exercise3/example-sync$ sudo ./deadlock
[sudo] password for ubuntu:
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
```

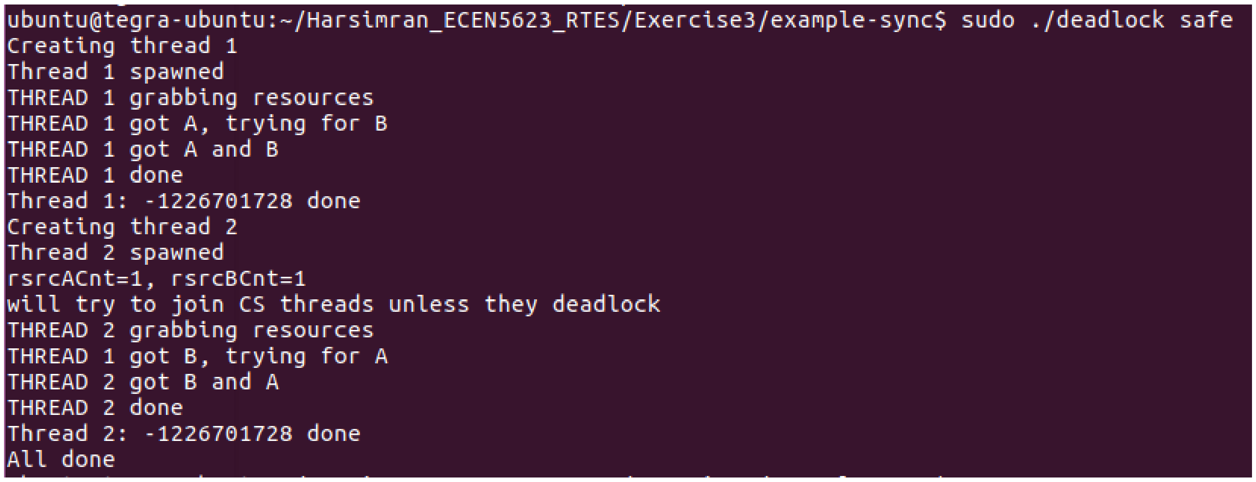
Here Thread 1 and Thread 2 are two tasks which are created in the main loop. Both the threads share rsrcA and rsrcB. In the unsafe condition thread1 and thread2 are spawned one after the other and both the threads are put to sleep after grabbing one of the resources. Because of this thread1 captures resource rsrcA and goes to sleep for 1 second. In the meantime thread2 is spawned and it captures rsrcB and goes to sleep for 1 second. When both the threads wake up, both the threads are blocked by each other as each thread has captured one of the resource and is waitng for the other one. This scenario is called deadlock and now the processor utilization becomes 0 as all both the processes are blocked.

The second scenario which is labeled as deadlock safe condition in the code that is when the code is executed using the following command

sudo ./deadlock safe

We get the following output

Deadlock safe

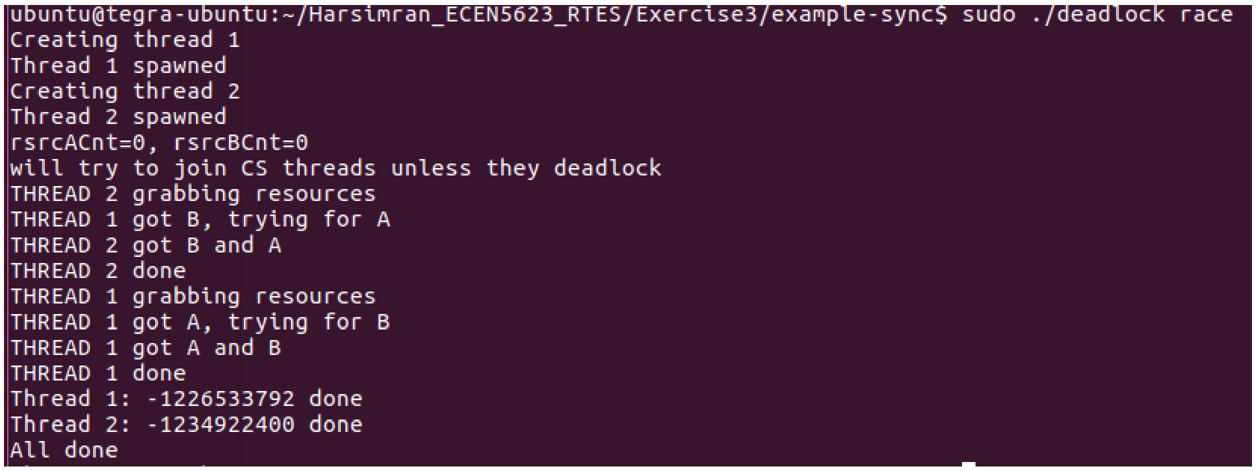


```
ubuntu@tegra-ubuntu:~/Harsimran_ECEN5623_RTES/Exercise3/example-sync$ sudo ./deadlock safe
Creating thread 1
Thread 1 spawned
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: -1226701728 done
Creating thread 2
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=1
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: -1226701728 done
All done
```

Here the code is executed in a safe condition. Thread1 is spawned in main loop first it is completed and terminated and then Thread2 is spawned. This completely eliminates the possibility of a deadlock situation. The flow of code is quite simple, first thread1 is created it captures rsrcA and sleeps for 1 second then it captures rsrcB and completes its execution. Then it is terminated using pthreadjoin command. After this Thread2 is created it captures both the resources and completes its execution. But this solution generally cannot be implemented in real time systems as any task can request for service at anytime and sequential execution cannot be implemented.

The third scenario which is labeled as deadlock race condition in the code that is when the code is executed using the following command
sudo ./deadlock race

Deadlock Race

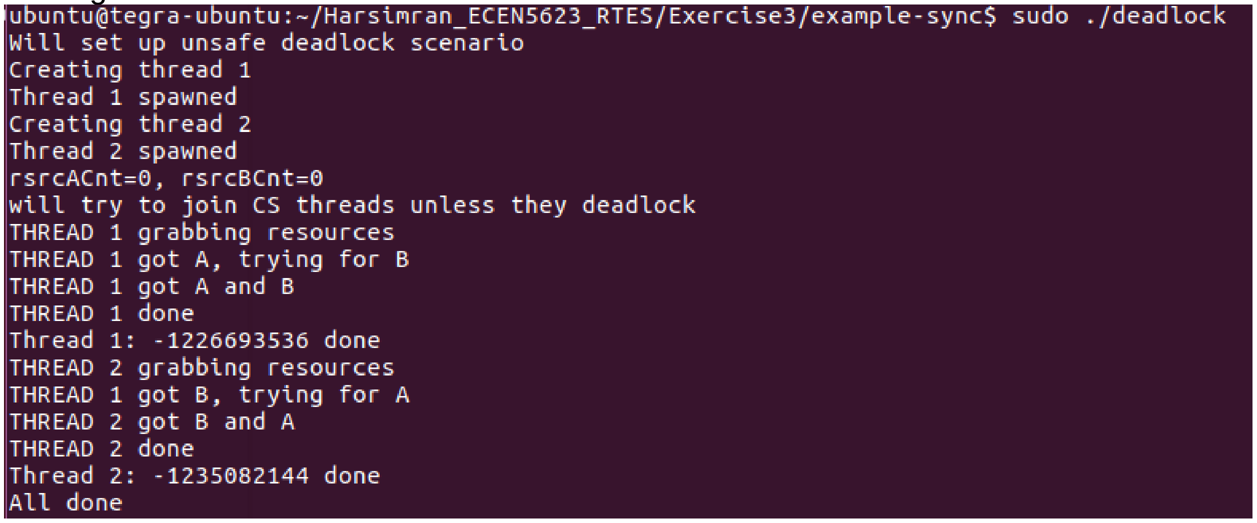


```
ubuntu@tegra-ubuntu:~/Harsimran_ECEN5623_RTES/Exercise3/example-sync$ sudo ./deadlock race
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
rsrcACnt=0, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 got B, trying for A
THREAD 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: -1226533792 done
Thread 2: -1234922400 done
All done
```

Here both the threads are in race condition. Thread1 and Thread2 are spawned one after the other and are in a race to capture both the shared resources. This can sometimes lead to deadlock but not always. Any thread can capture the shared resources first and completes its execution then the other thread captures the resources and completes its execution.

Finally when we fix the code which runs in unsafe condition the output of the code can be seen below

Making deadlock unsafe code work



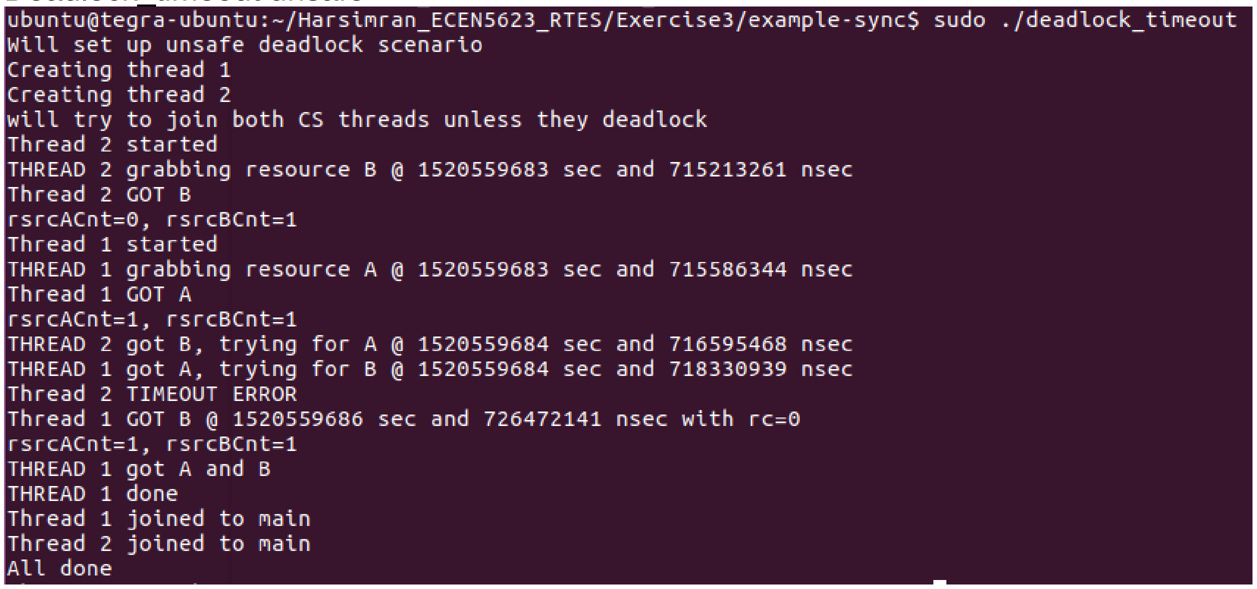
```
ubuntu@tegra-ubuntu:~/Harsimran_ECEN5623_RTES/Exercise3/example-sync$ sudo ./deadlock
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
rsrcACnt=0, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: -1226693536 done
THREAD 2 grabbing resources
THREAD 1 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
Thread 2: -1235082144 done
All done
```

The simplest way to make the unsafe code to work is to make thread2 sleep after it is spawned so that thread1 could capture both the resources and complete its execution and the case of deadlock can be avoided

Deadlock_timeout code description

This code has the same three cases which were present in the deadlock but with a major difference. The output of the deadlock_timeout code can be seen below

Deadlock_timeout unsafe



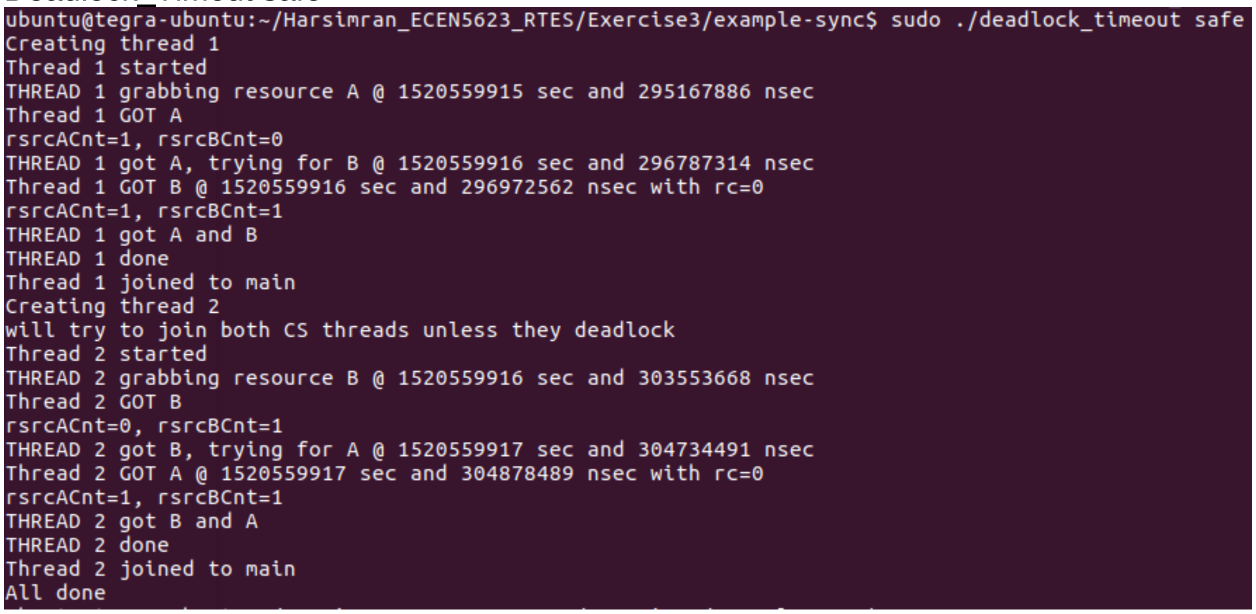
```
ubuntu@tegra-ubuntu:~/Harsimran_ECEN5623_RTES/Exercise3/example-sync$ sudo ./deadlock_timeout
Will set up unsafe deadlock scenario
Creating thread 1
Creating thread 2
will try to join both CS threads unless they deadlock
Thread 2 started
THREAD 2 grabbing resource B @ 1520559683 sec and 715213261 nsec
Thread 2 GOT B
rsrcACnt=0, rsrcBCnt=1
Thread 1 started
THREAD 1 grabbing resource A @ 1520559683 sec and 715586344 nsec
Thread 1 GOT A
rsrcACnt=1, rsrcBCnt=1
THREAD 2 got B, trying for A @ 1520559684 sec and 716595468 nsec
THREAD 1 got A, trying for B @ 1520559684 sec and 718330939 nsec
Thread 2 TIMEOUT ERROR
Thread 1 GOT B @ 1520559686 sec and 726472141 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
```

Here Thread 1 and Thread 2 are two tasks which are created in the main loop. Both the threads share rsrcA and rsrcB. In the unsafe condition thread1 and thread2 are spawned one after the other and both

the threads are put to sleep after grabbing one of the resources. Because of this thread1 captures resource rsrcA and goes to sleep for 1 second. In the meantime thread2 is spawned and it captures rsrcB and goes to sleep for 1 second. The main difference in this code and deadlock code is that there is timeout associated with each mutex lock. This basically means that if resource is not captured before the timeout then the thread is terminated and it unlocks all the resources it has captured. This breaks the deadlock situation. The result can be seen in the above output where thread2 timeout occurs when it tries to capture rsrcA and it gives up rsrcB and gets terminated. rsrcB is then captured by thread1. Hence the program doesn't remain in deadlock condition for indefinite amount of time. The issue with this approach is that thread2 gets terminated and never gets to execute which may not be acceptable in a real-time embedded system. The code to lock mutex with timeout is given below
pthread_mutex_timedlock();

The second scenario which is labeled as deadlock safe condition in the code that is when the code is executed using the following command
sudo ./deadlock_timeout safe
We get the following output

Deadlock_Timout safe

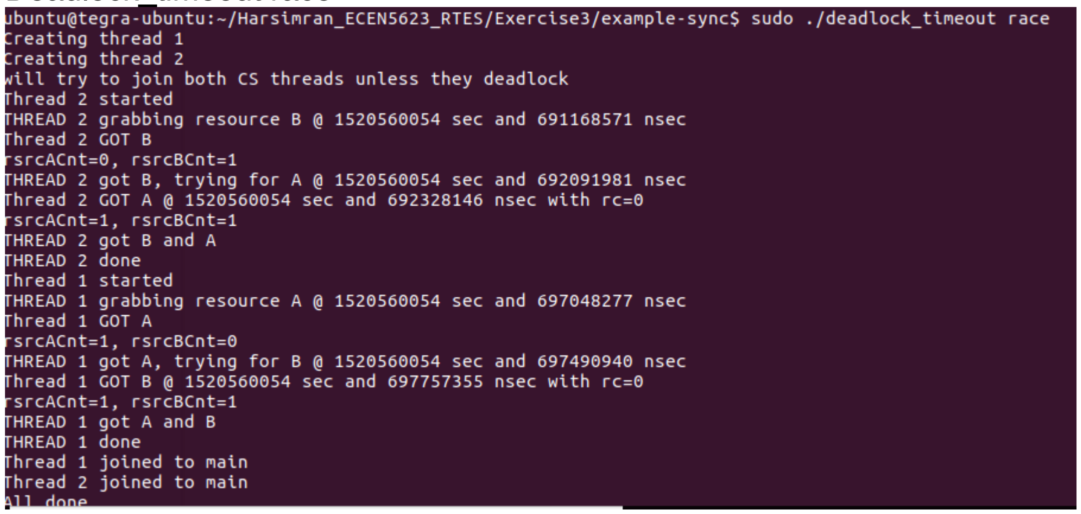


```
ubuntu@tegra-ubuntu:~/Harsimran_ECEN5623_RTES/Exercise3/example-sync$ sudo ./deadlock_timeout safe
Creating thread 1
Thread 1 started
THREAD 1 grabbing resource A @ 1520559915 sec and 295167886 nsec
Thread 1 GOT A
rsrcACnt=1, rsrcBCnt=0
THREAD 1 got A, trying for B @ 1520559916 sec and 296787314 nsec
Thread 1 GOT B @ 1520559916 sec and 296972562 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
THREAD 1 got A and B
THREAD 1 done
Thread 1 joined to main
Creating thread 2
will try to join both CS threads unless they deadlock
Thread 2 started
THREAD 2 grabbing resource B @ 1520559916 sec and 303553668 nsec
Thread 2 GOT B
rsrcACnt=0, rsrcBCnt=1
THREAD 2 got B, trying for A @ 1520559917 sec and 304734491 nsec
Thread 2 GOT A @ 1520559917 sec and 304878489 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
THREAD 2 got B and A
THREAD 2 done
Thread 2 joined to main
All done
```

This code runs the same way the deadlock code runs i.e Thread1 is spawned in main loop first it is completed and terminated and then Thread2 is spawned. This completely eliminates the possibility of a deadlock situation. The flow of code is quite simple, first thread1 is created it captures rsrcA and sleeps for 1 second then it captures rsrcB and completes its execution. Then it is terminated using pthreadjoin command. After this Thread2 is created it captures both the resources and completes its execution. But this solution generally cannot be implemented in real time systems as any task can request for service at anytime and sequential execution cannot be implemented. The only difference in this code is that each mutex lock is associated with timeout which is not of much use in this case.

The third scenario which is labeled as deadlock timeout race condition in the code that is when the code is executed using the following command
sudo ./deadlock_timeout race

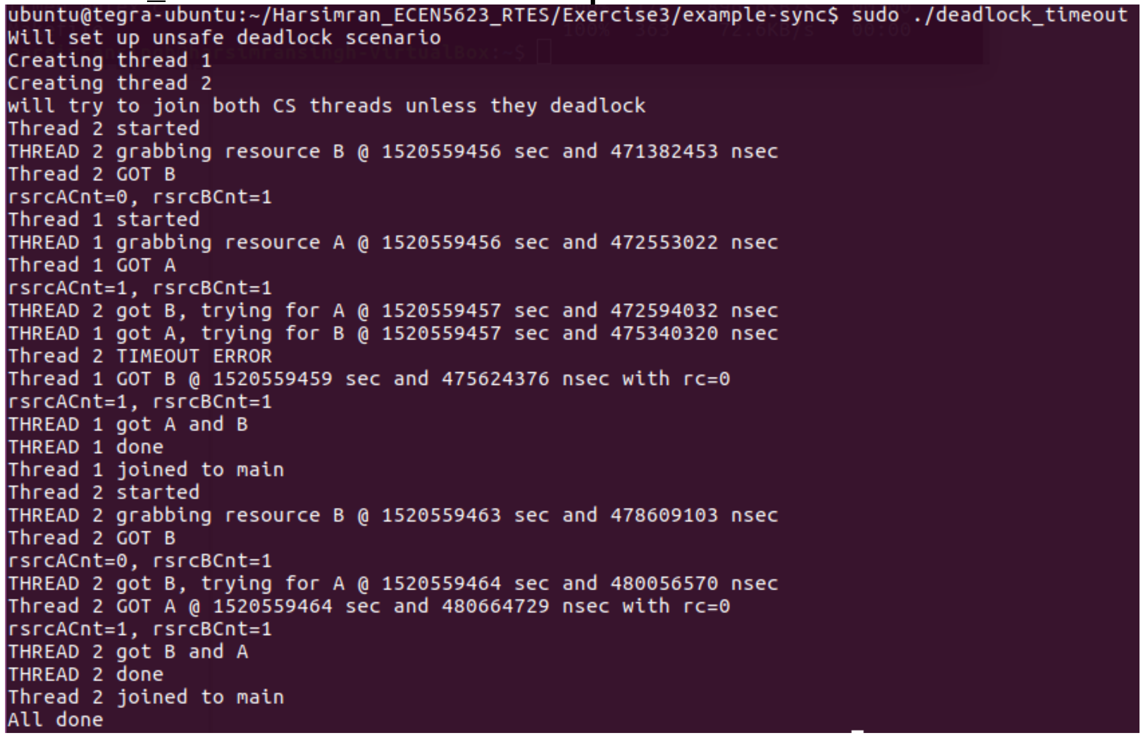
Deadlock_timeout race



```
ubuntu@tegra-ubuntu:~/Harsimran_ECEN5623_RTES/Exercise3/example-sync$ sudo ./deadlock_timeout race
Creating thread 1
Creating thread 2
will try to join both CS threads unless they deadlock
Thread 2 started
THREAD 2 grabbing resource B @ 1520560054 sec and 691168571 nsec
Thread 2 GOT B
rsrcACnt=0, rsrcBCnt=1
THREAD 2 got B, trying for A @ 1520560054 sec and 692091981 nsec
Thread 2 GOT A @ 1520560054 sec and 692328146 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
THREAD 2 got B and A
THREAD 2 done
Thread 1 started
THREAD 1 grabbing resource A @ 1520560054 sec and 697048277 nsec
Thread 1 GOT A
rsrcACnt=1, rsrcBCnt=0
THREAD 1 got A, trying for B @ 1520560054 sec and 697490940 nsec
Thread 1 GOT B @ 1520560054 sec and 697757355 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
```

Here both the threads are in race condition. Thread1 and Thread2 are spawned one after the other and are in a race to capture both the shared resources. This can sometimes lead to deadlock but not always. Any thread can capture the shared resources first and completes its execution then the other thread captures the resources and completes its execution. The only difference in this code is that each mutex lock is associated with timeout which is not of much use in this case.

Deadlock_timeout corrected Code Output



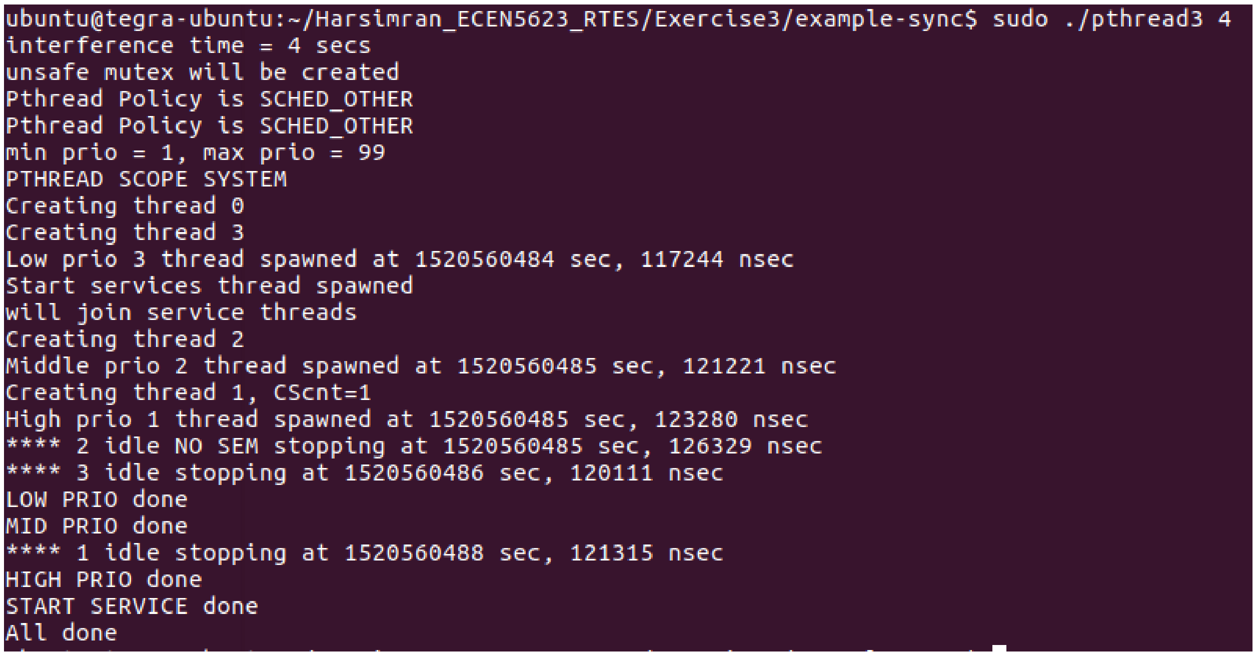
```
ubuntu@tegra-ubuntu:~/Harsimran_ECEN5623_RTES/Exercise3/example-sync$ sudo ./deadlock_timeout
Will set up unsafe deadlock scenario
Creating thread 1
Creating thread 2
will try to join both CS threads unless they deadlock
Thread 2 started
THREAD 2 grabbing resource B @ 1520559456 sec and 471382453 nsec
Thread 2 GOT B
rsrcACnt=0, rsrcBCnt=1
Thread 1 started
THREAD 1 grabbing resource A @ 1520559456 sec and 472553022 nsec
Thread 1 GOT A
rsrcACnt=1, rsrcBCnt=1
THREAD 2 got B, trying for A @ 1520559457 sec and 472594032 nsec
THREAD 1 got A, trying for B @ 1520559457 sec and 475340320 nsec
Thread 2 TIMEOUT ERROR
Thread 1 GOT B @ 1520559459 sec and 475624376 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
THREAD 1 got A and B
THREAD 1 done
Thread 1 joined to main
Thread 2 started
THREAD 2 grabbing resource B @ 1520559463 sec and 478609103 nsec
Thread 2 GOT B
rsrcACnt=0, rsrcBCnt=1
THREAD 2 got B, trying for A @ 1520559464 sec and 480056570 nsec
Thread 2 GOT A @ 1520559464 sec and 480664729 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
THREAD 2 got B and A
THREAD 2 done
Thread 2 joined to main
All done
```

The issue with initial deadlock_timeout was thread2 was terminated after timeout. In the revised code we use random back off along with timelocked mutex to get out of the deadlock. We comment out the pthread_exit() command in the timeout error loop. We use random sleep using rand() function for random back off.

Pthread code

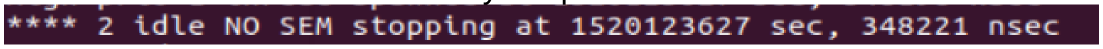
This code displays how the code will execute if there is a case of priority inversion in the system. Priority inversion can be simply defined as any time when a high-priority service is put in a wait state while a lower-priority service gets to run, this can occur in any blocking scenario.[4]

Pthread3 code screenshot



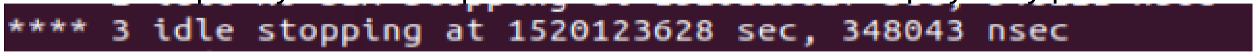
```
ubuntu@tegra-ubuntu:~/Harsimran_ECEN5623_RTES/Exercise3/example-sync$ sudo ./pthread3 4
interference time = 4 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Creating thread 3
Low prio 3 thread spawned at 1520560484 sec, 117244 nsec
Start services thread spawned
will join service threads
Creating thread 2
Middle prio 2 thread spawned at 1520560485 sec, 121221 nsec
Creating thread 1, CScnt=1
High prio 1 thread spawned at 1520560485 sec, 123280 nsec
**** 2 idle NO SEM stopping at 1520560485 sec, 126329 nsec
**** 3 idle stopping at 1520560486 sec, 120111 nsec
LOW PRIO done
MID PRIO done
**** 1 idle stopping at 1520560488 sec, 121315 nsec
HIGH PRIO done
START SERVICE done
All done
```

Initially a the interference time is taken during run time from the user. Then the start service thread is spawned in the main loop. In the start service thread there are three threads that are spawned with high(Thread 1), medium(Thread 2) and low(Thread 3) priority. A resource which is protected by semaphore msgsem is shared between the high and low priority thread. Medium priority thread doesn't share any resource with other threads. Initially low priority thread is spawned and then the medium and high priority threads are spawned. Low priority thread captures the shared resource and blocks high priority thread from executing. Since medium priority thread does not have any shared resource it can preempt low priority thread and complete its execution. Hence first the medium priority thread completes its execution which can be seen by the printf



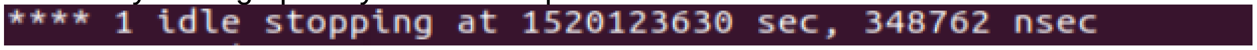
```
**** 2 idle NO SEM stopping at 1520123627 sec, 348221 nsec
```

After this low priority thread completes its execution which is displayed by printf



```
**** 3 idle stopping at 1520123628 sec, 348043 nsec
```

and finally the high priority thread completes its execution



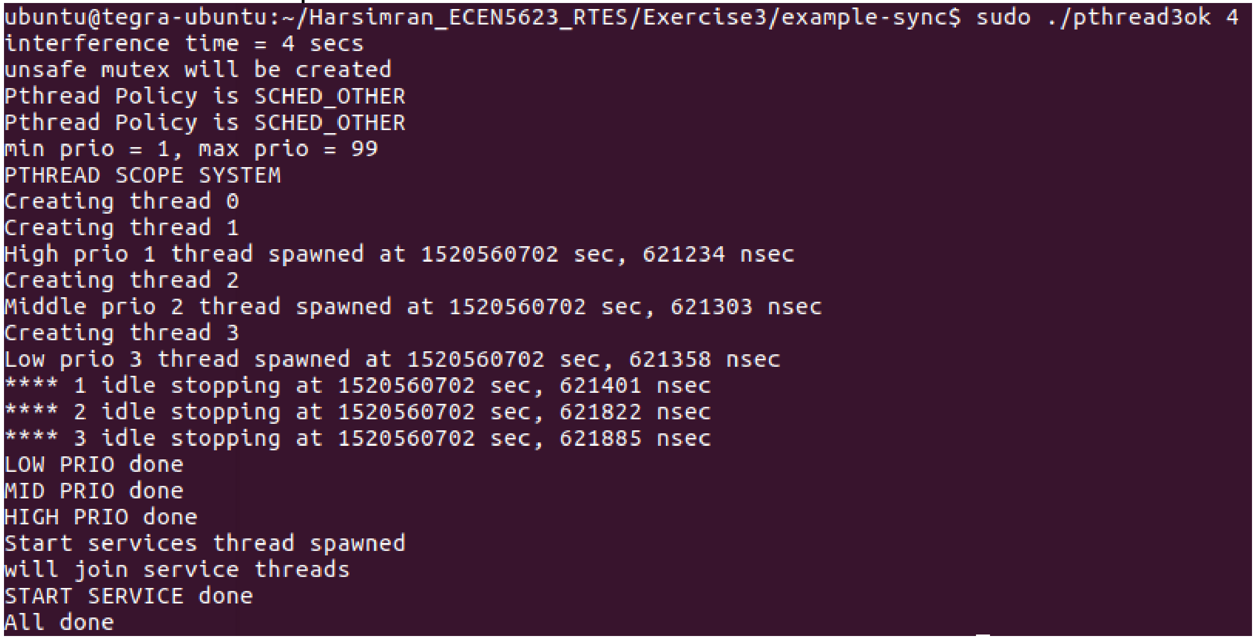
```
**** 1 idle stopping at 1520123630 sec, 348762 nsec
```

. This scenario when lower priority tasks complete before the high priority task i.e high priority task is blocked by low priority task which is preempted by medium priority task is a case of unbounded priority inversion.

Pthread3ok

The issue of unbounded priority inversion is solved in the pthread3ok code. Here it eliminates the use of mutex hence High priority service can preempt the low priority service and the issue of priority inheritance is avoided. But this is not a good solution as there are some cases where use of synchronisation primitives are unavoidable.

Pthread3ok code output

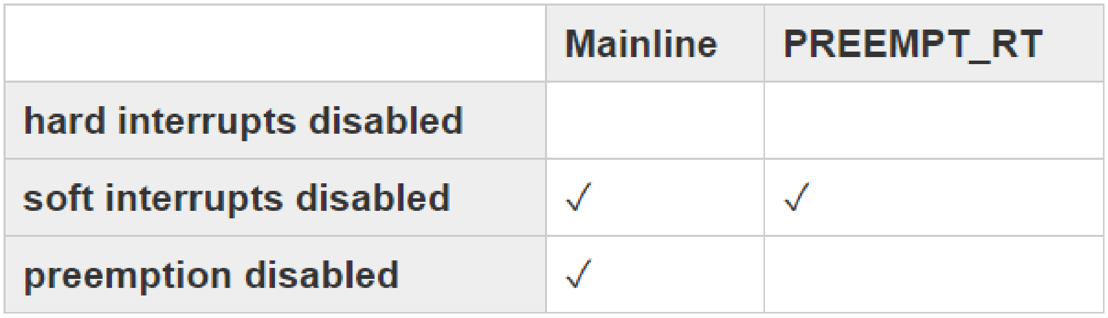


```
ubuntu@tegra-ubuntu:~/Harsimran_ECEN5623_RTES/Exercise3/example-sync$ sudo ./pthread3ok 4
interference time = 4 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Creating thread 1
High prio 1 thread spawned at 1520560702 sec, 621234 nsec
Creating thread 2
Middle prio 2 thread spawned at 1520560702 sec, 621303 nsec
Creating thread 3
Low prio 3 thread spawned at 1520560702 sec, 621358 nsec
**** 1 idle stopping at 1520560702 sec, 621401 nsec
**** 2 idle stopping at 1520560702 sec, 621822 nsec
**** 3 idle stopping at 1520560702 sec, 621885 nsec
LOW PRIO done
MID PRIO done
HIGH PRIO done
Start services thread spawned
will join service threads
START SERVICE done
All done
```

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

According to me RT_PREEMPT_PATCH can make Linux real-time safe upto a certain extent due to following reasons

- It uses high precision timer for real time scheduling initially linux used to use jiffys.
- In mainline linux the ISR is processed in hardware interrupt context with hardware interrupts disabled. Hardware interrupts disabled further implies that the software interrupts and preemptions are also disabled. The kernel command line option `threadirqs` changes the the interrupt handlers to run in a threaded context. The scheduler policy of the thread is set to SCHED_FIFO with a default priority of 50. The RT preempt patch forces this threadirqs command option which doesn’t allow the to disable preemption.



	Mainline	PREEMPT_RT
hard interrupts disabled		
soft interrupts disabled	✓	✓
preemption disabled	✓	

- Spin locks are generally used in kernel space to protect the critical section. Its advantage is its speed of execution but when locks are captured for a long time spin locks hamper the performance of the system and increases latency. Because of this reason RT_PREEMPT_PATCH converts most of the kernel space spin locks to rt_mutex locks
- It implements priority inheritance for in-kernel spinlocks and semaphores.

Question 4

Review heap_mq.c and posix_mq.c. First, re-write the VxWorks code so that it uses RT-Linux Pthreads (FIFO) instead of VxWorks tasks, and then write a brief paragraph describing how these two message queue applications are similar and how they are different. You may find the following Linux POSIX demo code useful, but make sure you not only read and port the code, but that you build it, load it, and execute it to make sure you understand how both applications work and prove that you got the POSIX message queue features working in Linux on your Altera DE1-SoC or Jetson. Message queues are often suggested as a way to avoid MUTEX priority inversion. Would you agree that this really circumvents the problem of unbounded inversion? If so why, if not, why not?

Solution:

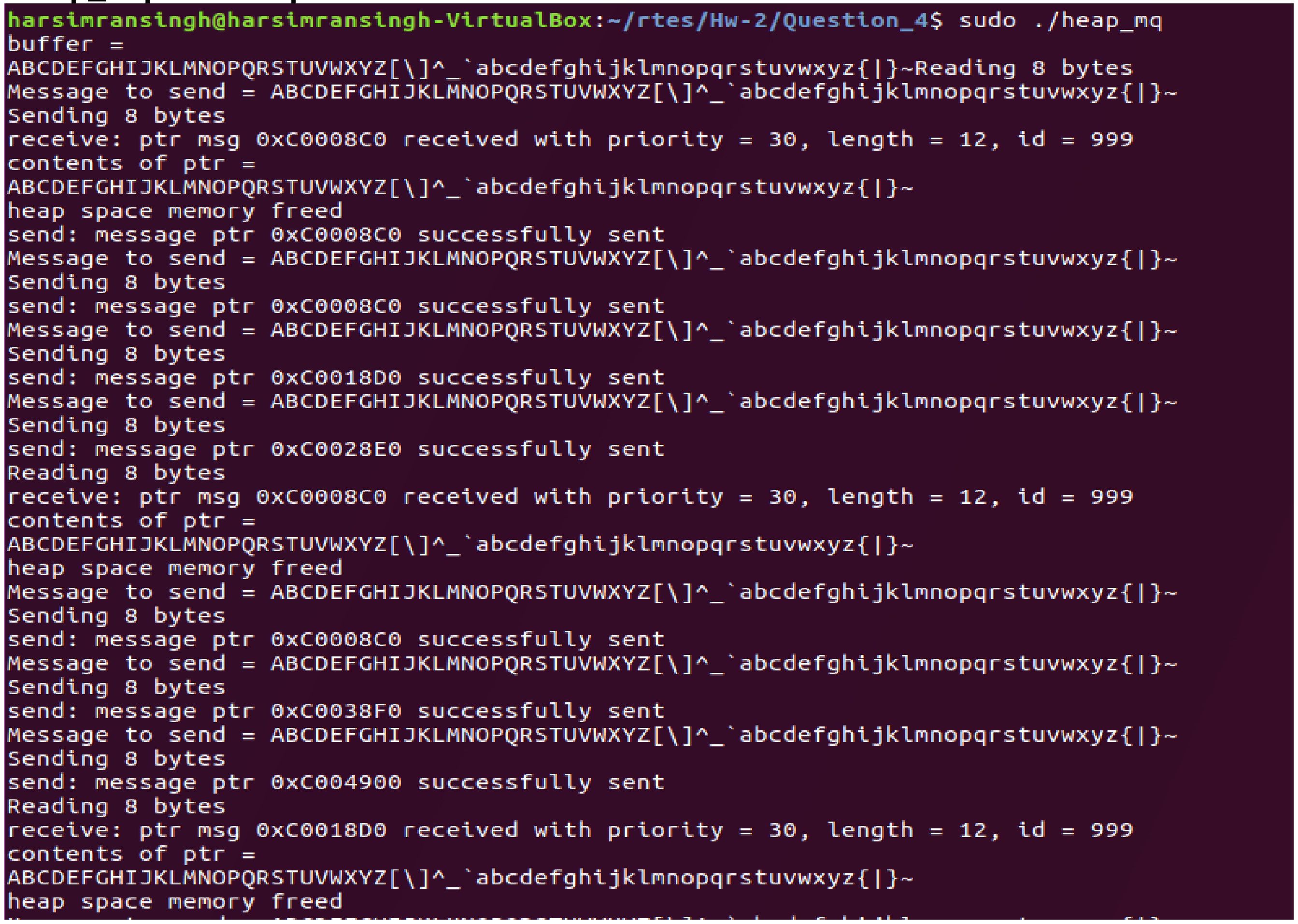
Differences

Points	Heap_mq	Posix_mq
Message le	Only the pointer to the message and ID value is sent using the mq_send	Whole message is sent using the mq_send command.
Memory	More memory is required since the message is first copied in heap memory buffptr and then the pointer to this memory address is sent using my_send if there is only one message. But as number of messages increase this code becomes more efficient since same heap space is used again and again to transfer messages. And the files can directly be copied to heap using pointers before copying it to stack.	Complete message is sent using mq_send, so there is no need to copy it in heap memory again. But as number of messages to send increases code will occupy more memory as the messages are first brought into stack and then complete message is sent.
Execution	Here the task keeps on running until we exit it manually. Two tasks are spawned with receiver having the higher priority 90 and sender having lower priority 100.	Here the code runs only once. The receiver is given higher priority but mq_receive is blocking so sender thread runs and transmits the message which is then received by the receiver thread.

Similarities

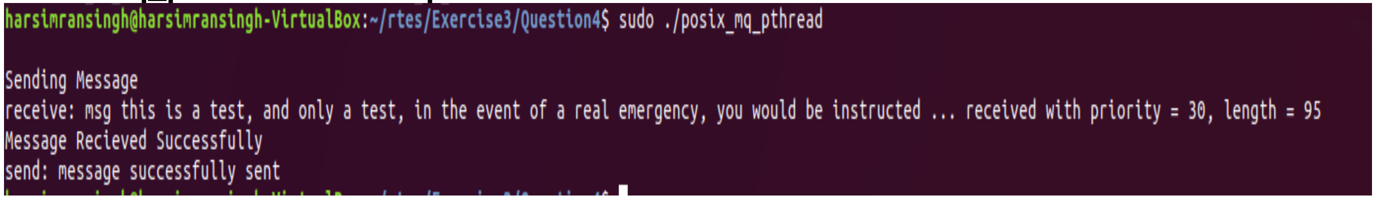
- Both the codes use same message queue API to send and receive message.
- Both the codes use the same philosophy of reading the oldest highest priority message first from the queue.
- Both the codes check for error after each stage and print it using the perror command.

Heap_mq code output



```
harsimransingh@harsimransingh-VirtualBox:~/rtes/Hw-2/Question_4$ sudo ./heap_mq
buffer =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~Reading 8 bytes
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
receive: ptr msg 0xC0008C0 received with priority = 30, length = 12, id = 999
contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
heap space memory freed
send: message ptr 0xC0008C0 successfully sent
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
send: message ptr 0xC0008C0 successfully sent
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
send: message ptr 0xC0018D0 successfully sent
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
send: message ptr 0xC0028E0 successfully sent
Reading 8 bytes
receive: ptr msg 0xC0008C0 received with priority = 30, length = 12, id = 999
contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
heap space memory freed
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
send: message ptr 0xC0008C0 successfully sent
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
send: message ptr 0xC0038F0 successfully sent
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
send: message ptr 0xC004900 successfully sent
Reading 8 bytes
receive: ptr msg 0xC0018D0 received with priority = 30, length = 12, id = 999
contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
heap space memory freed
```

Posix-mq_pthread code output



```
harsimransingh@harsimransingh-VirtualBox:~/rtes/Exercise3/Question4$ sudo ./posix_mq_pthread

Sending Message
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
Message Recieved Successfully
send: message successfully sent
```

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

The most common and prominent issue associated with global memory sharing is priority inversion. Priority inversion can basically be defined as a scenario when the higher priority task is blocked by a lower priority task due to lock on a shared resource. Locks are generally implemented using synchronisation primitives which causes priority inversion. The priority inversion becomes unbounded when a medium priority task keeps on preempting the low priority task and therefore high priority task is block for indefinite amount of time. The issues associated with global memory sharing can be solved by using inversion safe mutex.

Message queues are basically used to synchronise tasks by passing messages. mq_recieve() command is a blocking type of command i.e. if the queue is empty it waits for the other task to add a message to the message buffer. For this message queues use shared memory for passing on the messages and this shared memory is protected by using locks which is implemented by the kernel.

The implementation of message queue depends on the kernel, which means that whether message queue would be able to solve issues like unbounded priority inversion would depend upon the operating system used. Vxworks have priority inversion safe mutexes implemented in message queues hence it would solve the problem of unbounded priority inversion. Similarly most of the other modern OS are able to solve the issue of unbounded priority. Even the POSIX message queues use the concept of message priority which basically means that messages with higher priority would be dequeued before messages with lower priority. Hence message queues solve the issues associated with global sharing in this case too. Basically According to us, If the message queue implementation by a particular kernel uses priority inversion safe mutexes then it can solve the issues associated with global memory sharing.

Question5:
Watchdog timers, timeouts and timer services – First, read this overview of the Linux Watchdog Daemon and describe how it might be used if software caused an indefinite deadlock. Next, to explore timeouts, use your code from #2 and create a thread that waits on a MUTEX semaphore for up to 10 seconds and then un-blocks and prints out "No new data available at <time>" and then loops back to wait for a data update again. Use a variant of the pthread_mutex_lock called pthread_mutex_timedlock to solve this programming problem.

Solution:
A watchdog timer is an electronic timer that is used to detect and recover from computer malfunctions. During normal operation, the computer regularly resets the watchdog timer to prevent it from elapsing, or "timing out". If, due to a hardware fault or program error, the computer fails to reset the watchdog, the timer will elapse and generate a timeout signal. The timeout signal is used to initiate corrective action or actions. The corrective actions typically include placing the computer system in a safe state and restoring normal system operation.

The Watchdog in LINUX operating system has two parts in it:
1. The actual hardware timer and a kernel driver module that can force a hard reset, and;
2. The user-space background daemon that refreshes the timer and provides a wider range of health monitoring and recovery options.

Both can function independently, but clearly, they are designed to operate together for maximum protection.

The Watchdog Module (Hardware Timer):
Normally, the hardware support for a watchdog is simply a timer that is set to some reasonable time-out, and then periodically refreshed by the running software. If for any reason the software stops refreshing the hardware then it times-out and performs a hardware reset of the computer. In this way, even the kernel panic type of faults can usually be recovered.

In Linux operating system, there is a standard interface to the watchdog hardware provided by the corresponding kernel device driver (module) provided as **/dev/watchdog**. To load this driver, we need to add the module name to **/etc/modules** or to **/etc/default/watchdog** by editing watchdog_module="none" to have the module name. The watchdog hardware + driver module provides the most basic of protection. It is started by anything that can periodically write to **/dev/watchdog** and if that fails for any reason the watchdog hardware times-out and machine is rebooted by means of a hard reset.

The Watchdog Daemon:
To operate the watchdog device, there is normally a background daemon that can open the device and provide the periodic refresh activity. However, a machine can also get in to a very unusable state without terminating the background daemon's operation, therefore the watchdog daemon for Linux can be configured to periodically run many basic tests to verify that the machine is working fine.

The daemon ensures safety by either of these two methods namely “moderately orderly” shutdown or “the blunderbuss approach”. First the daemon would try to perform the reboot in a moderately orderly manner, executing the init based shutdown scripts while trying to keep the log of the issue. If this approach fails the daemon resorts to “the blunderbuss approach” killing all the system process, syncing CMOS clocks and syncing and unmounting filesystems and resets the system with the hardware timer.

<u>Description of how WD timer be used in recovering from software caused deadlock:</u>

Suppose there are two independent processes P1 and P2 sharing resources in their critical sections. In normal operation of the system the watchdog timer is enabled and will get reset by both the processes periodically.

Unfortunately for any reason, if they run into a deadlock i.e. both the processes are blocked on each-other waiting indefinitely for the shared resource. Now, either of the processes will fail to do a timer reset on the watchdog, which would eventually run out and perform a hard reset of the system. This would do two things, firstly it would resume the system back from deadlock and also inform the user about the glitch occurred in their program so that it can be corrected.

There is another scenario in which the WD timer can help i.e If user runs a process which has a glitch and takes up most of the system resources leaving the system operational & still refreshing the hardware timer. In this case, if the watchdog daemon is enabled it would detect this and perform reset to resume the system to normal operation.

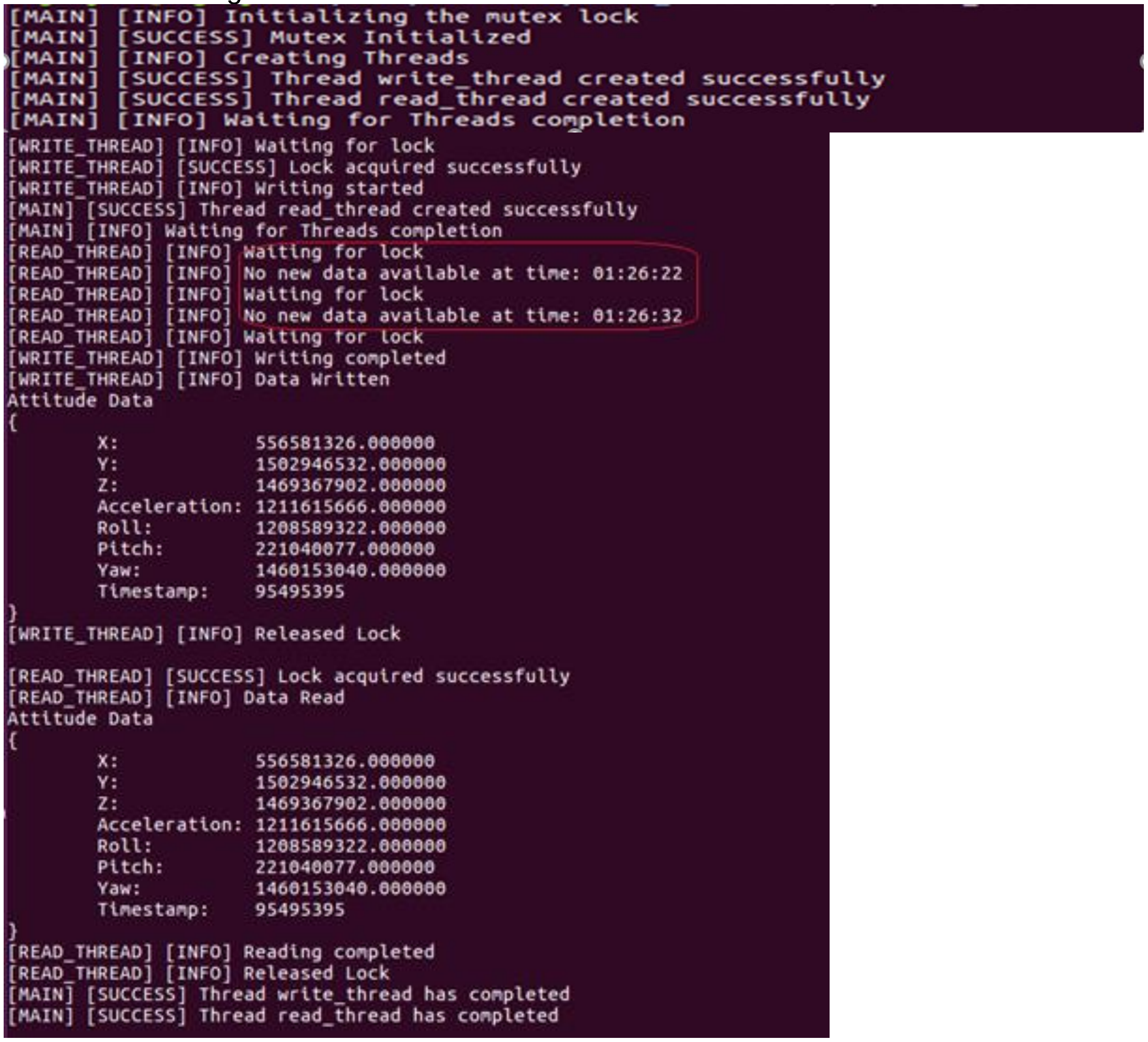
<u>Adaption of code from #2 to handle Timeouts:</u>

The code from question 2 is modified to handle timeouts for accessing the shared resource.

The implemented code has two child threads created by the parent thread(main). One child thread (write_thread) is writing the attitude data and the other child thread (read_thread) is reading same data written by first thread. To protect the shared data from being accessed by both threads simultaneously, we implemented mutex lock protection using function pthread_mutex_lock() & pthread_mutex_timedlock(). Now, write_thread acquires the mutex and sleeps for seconds (more than the timeout 10s, to see the printing of “No new data is available at <time>”) to write the data. And, the read_thread will try to acquire the mutex lock using pthread_mutex_lock() and expires after every 10s (expires two times in 21 seconds) which can be seen in the below figure. Whenever it expires without getting mutex clock “No new data available at <time>” message is displayed. The next time when the read_thread tries and after the write thread has released the lock on the shared resource (indicating its work is done), then the reader thread gets the lock and reads the data. In this way, we have demonstrated the shared global resource is accessed by both threads without corruption while handling timeouts.

Output screen shot:

write_thread sleeps for 21 seconds and so timeout (10 seconds) expires 2 times in read_thread as shown in below figure:



```
[MAIN] [INFO] Initializing the mutex lock
[MAIN] [SUCCESS] Mutex Initialized
[MAIN] [INFO] Creating Threads
[MAIN] [SUCCESS] Thread write_thread created successfully
[MAIN] [SUCCESS] Thread read_thread created successfully
[MAIN] [INFO] Waiting for Threads completion
[WRITE_THREAD] [INFO] Waiting for lock
[WRITE_THREAD] [SUCCESS] Lock acquired successfully
[WRITE_THREAD] [INFO] Writing started
[MAIN] [SUCCESS] Thread read_thread created successfully
[MAIN] [INFO] Waiting for Threads completion
[READ_THREAD] [INFO] Waiting for lock
[READ_THREAD] [INFO] No new data available at time: 01:26:22
[READ_THREAD] [INFO] Waiting for lock
[READ_THREAD] [INFO] No new data available at time: 01:26:32
[READ_THREAD] [INFO] Waiting for lock
[WRITE_THREAD] [INFO] Writing completed
[WRITE_THREAD] [INFO] Data Written
Attitude Data
{
        X:            556581326.000000
        Y:            1502946532.000000
        Z:            1469367902.000000
        Acceleration: 1211615666.000000
        Roll:         1208589322.000000
        Pitch:        221040077.000000
        Yaw:          1460153040.000000
        Timestamp:    95495395
}
[WRITE_THREAD] [INFO] Released Lock

[READ_THREAD] [SUCCESS] Lock acquired successfully
[READ_THREAD] [INFO] Data Read
Attitude Data
{
        X:            556581326.000000
        Y:            1502946532.000000
        Z:            1469367902.000000
        Acceleration: 1211615666.000000
        Roll:         1208589322.000000
        Pitch:        221040077.000000
        Yaw:          1460153040.000000
        Timestamp:    95495395
}
[READ_THREAD] [INFO] Reading completed
[READ_THREAD] [INFO] Released Lock
[MAIN] [SUCCESS] Thread write_thread has completed
[MAIN] [SUCCESS] Thread read_thread has completed
```

References

1. https://lwn.net/Articles/177111/ - Ingo Molnar PI-Futex Patch
2. https://en.wikipedia.org/wiki/Futex - futex wikipedia page
3. https://www.akkadia.org/drepper/futex.pdf - futex are tricky by Ulrich Drepper
4. Real-Time Embedded Components and Systems with Linux and Rtos By Dr Sam Siewert and John Pratt
5. https://wiki.linuxfoundation.org/realtime/documentation/start - RT_PREEMPT Patch
6. https://en.wikipedia.org/wiki/Reentrancy_(computing) -Reentrancy wikipedia
7. https://en.wikipedia.org/wiki/Thread_safety - Thread safety wikipedia
8. http://www.sat.dundee.ac.uk/psc/watchdog/watchdog-background.html -Watchdog timer overview

Exercise staff

Harsimransingh Bindra	Nagarjuna Reddy
Graduate Student Embedded Systems Engineering University Of Colorado Boulder Harsimransingh.Bindra@colorado.edu	Graduate Student Embedded Systems Engineering University Of Colorado Boulder Nagarjuna.Reddy@colorado.edu