



Exercise #3 – Threading/Tasking and Real-Time Synchronization



SOWMYA RAMAKRISHNAN
VIKRANT WAJE

ECEN 5623, Real-Time Systems:

Exercise #3 – Threading/Tasking and Real-Time Synchronization

DUE: As Indicated on D2L and in class

Please thoroughly read Chapters 6, 7 & 8 in the text.

Please see example code provided - <http://mercury.pr.erau.edu/~siewerts/cec450/code/>

Please complete this lab on an Altera DE1-SoC or Jetson and provide evidence that you did your work with screenshots.

Exercise #3 Requirements:

- 1) [10 points] Read Sha, Rajkumar, et al paper, "[Priority Inheritance Protocols: An Approach to Real-Time Synchronization](#)" and summarize 3 main key points the paper makes. Read [Dr. Siewert's summary paper on the topic as well](#). Finally, read the positions of [Linux Torvalds as described by Jonathan Corbet](#) and [Ingo Molnar and Thomas Gleixner](#) on this topic as well. Take a position on this topic yourself and write at least one well supported paragraph or more to defend your position based on what we have learned in class. Does the PI-futex ([Futex](#), [Futexes are Tricky](#)) that is described by Ingo Molnar provide safe and accurate protection from un-bounded priority inversion as described in the paper? If not, what is different about it?

ANSWER

The paper "Priority Inheritance Protocols: An Approach to Real-Time Synchronization" was read and the **main key points that it makes are summarized** as follows.

- For tasks in priority-driven pre-emptive real-time systems, when semaphores are used for synchronization shared memory access, blocking issues can arise, with lower priority tasks blocking a task of high priority for an indefinite period of time. In order to solve this issue, priority inheritance techniques should be used.
- It can be seen from the understanding of the paper that a job can be blocked at most once per semaphore it uses. [Theorem 6] Thus, the higher priority task can be blocked for a maximum of 'm' times if there are 'm' semaphores. This point is key, since the blocking time for each semaphore must be bounded – assuming no job has infinite execution. This results in a job that has a finite upper bound on its blocking time.
- Priority inheritance lets the lower priority task, which has locked the semaphore, inherit the priority of the higher priority task such that it runs as the higher priority task. Thus, one task's priority is inherited by another. While this does solve the problem of

unbounded blocking, it still has issues in that it cannot prevent deadlocks. Also, the unbounded blocking period can have significance due to chain of blocking by other tasks.

- In order to solve the issues of deadlock and chained blocking, and to bound the blocking period, the priority ceiling protocol is used. Each task has a basic priority inheritance associated with it. Priority ceiling protocol assigns a priority ceiling to each semaphore. This results in an inherited higher priority task blocking a high priority task while sharing multiple semaphores such that the inherited task gets all the required semaphores. This effectively avoids deadlocks and chained blocking. Thus, a higher priority task can be blocked by a lower priority task for at most one critical section of all critical sections of the lower priority task. [Lemma 11]
- Another important note on the priority ceiling protocol is that there isn't any transitive blocking with this protocol. Thus, a low priority task will immediately be pushed to the highest priority of all tasks it can block for any given semaphore. [Lemma 9]

Linux Position on Priority Inheritance in the kernel and Why it makes sense [Our Position on this topic]

In the context of using priority inheritance to avoid priority inversion, our group is taking the position in favor of using it instead of not using it as mandated by Linus Torvalds. Linux Priority Inheritance is a good thing, and can be extremely useful, depending on the application.

Linus Torvalds takes the position that any priority inversion issue can be solved without priority inheritance, which, we feel, is unrealistic in a development point of view. For completely fair scheduling used in desktop OSes today, there is not any priority inversion as tasks are not preempted and all of them run for a fair share of time. But in the case of SCHED_FIFO or RTOS, higher priority tasks will pre-empt lower priority tasks which may lead to tasks wanting to access shared memory simultaneously, which in turn leads to lower priority tasks taking higher priority – the issue of priority inversion. The low priority task could potentially write to locations where they must not. As the number of processes using the shared memory increases, the probability of blocking increases and this unbounded blocking can easily cause system failure. The solution, thus, for such cases of priority inversion, is priority inheritance. In priority inheritance, lower priority tasks can use inherited priority to run as higher priority tasks. Deadlocks can also be prevented using Priority ceiling protocol. These methods, however, are complicated and would slow down the locking code. They would thus be rejected from being merged into the Linux kernel code. Also, one may argue that the inversion problem might be overcome by running code in user space, and never giving as high priority to a task that it over-writes kernel space memory. Other potential implementations would not even require Priority Inheritance. If blocking is not used, inversion would never happen, but that requires every function to be given a copy of memory that it works on, which again is not efficient practice. Linux is in the market of user-end OS and hence their position is vindicated. However, as far as hard real-time applications are concerned, Priority inheritance solves the problem of inversion. This is a solution that, despite its

drawbacks listed above, can be made to work better and more easily than other solutions, and hence the position is taken for using priority inheritance to solve priority inversion. Also, Linux is not essentially designed for Hard Real-time system applications.

Futexes as a solution for Unbounded Priority Inversion

Futexes are fast locking mechanisms used in user space. Igno Molnar developed priority inheritance support for futexes. The PI Futex prevents a high priority task from unbounded priority inversion in user space. By implementing PI Futexes largely the same as normal Futexes for the fast path, while not invoking any kernel calls for the short path, Igno Molnar made a safe implementation of Priority Inheritance Blocking. Also, the implementation for slow lock uses a minimal amount of kernel calls while looking up the priority of the highest waiting task.

Futexes are operated in user space and hence can easily be modified using an atomic operation like CMPXCHG in x86. Since this happens in user space, the kernel maintains no information about the lock state. The Native POSIX Thread Library (NPTL) uses PI-Futexes for Pthread MUTEX implementation in user space. They provide safe and accurate protection from unbounded priority inversion in user space. In user space, applications cannot use spinlocks to disable interrupts or make a task non pre-emptable in a critical section. Thus, there might still be unbounded inversion due to blocking of in-kernel MUTEXes, spinlocks or semaphores. In such cases, in order to ensure deterministic scheduling for applications in user space, priority inheritance has to be used. When there is no contention for shared memory access, the PI-Futex works faster compared to the kernel space mutex implementation as kernel level calls are absent. In the presence of a contention, the FUTEX_LOCK_PI operation is requested from the kernel. The kernel will then use priority inheritance protocols to carry out the rest of the task. Thus, while Futexes seem like a good implementation, they aren't in the kernel, and the kernel effectively knows about the PI-Futex only if there is contention.

2) [25 points] Review the terminology and describe clearly what it means to write "thread safe" functions that are "re-entrant". There are generally three main ways to do this: 1) pure functions that use only stack and have no global memory, 2) functions which use thread indexed global data, and 3) functions which use shared memory global data, but synchronize access to it using a MUTEX semaphore critical section wrapper. Describe each method and how you would code it and how it would impact real-time threads/tasks. Now, using a MUTEX, provide an example using RT-Linux Pthreads that does a thread safe update of a complex state with a timestamp ([pthread mutex lock](#)). Your code should include two threads and one should update a timespec structure contained in a structure that includes a double precision attitude state of {X,Y,Z acceleration and Roll, Pitch, Yaw rates at Sample_Time} (just make up values for the navigational state and see http://linux.die.net/man/3/clock_gettime for how to get a precision timestamp). The second thread should read the times-stamped state without the possibility of data corruption (partial update).

ANSWER

Description of methods

Three ways to write Thread-safe functions that are re-entrant

- The first way of creating thread-safe functions is to create pure functions that only use stack variables and have no global data or dynamic memory. This is the easiest way to ensure non-corruption of data, and this is achieved due to the absence of any shared data between two or more threads, each function having its own local data that can be modified without affecting others. Each thread working on the function will have a local copy of the variable unlike a global declaration which would result in a race condition. This method can hence be implemented by writing simple functions, each with their own local data. All data that every function needs must be passed to it, and the functions should return all processed data. It is necessary, though, to make sure that the threads must be executed in order, with proper priority assigned to each thread, to ensure no issues or problems with data forwarding. While this method is easy, it is cumbersome because it, more often than not, leads to over-sized structures getting passed in to functions unnecessarily. Copying all data every time a function call is made is extremely inefficient. Also, it does not make practical sense, since in most practical cases, sharing of data between real-time threads is required.
- The second method of implementation of a thread-safe function, while overcoming the limitations of the first, is to use functions which use thread-indexed global data. This ensures that the thread remains safe, while having global variables to share data. Since data is thread-indexed, either many copies of the same memory will have to be made, or proper locking mechanisms will have to be implemented to prevent shared memory errors. Thread-indexed global data defines a mechanism wherein the code can retrieve thread-specific data stored in a global database which can be accessed with the use of a shared key that is known to all threads involved in accessing the function. When a thread

requests data using the key, it will get an address back, which can be used to securely access the data. Thus, this method enables threads to communicate in real-time systems. The problem, however, is the synchronization between threads. If the threads need to share data, a shared memory has to be used with MUTEXes and semaphores. Priorities to each thread must be assigned with care. Transferring data from global memory for computation purposes will result in lesser throughput due to high latency.

- The final method is to use functions which use shared memory global data but synchronize access to it using a MUTEX semaphore critical section wrapper. It is essentially a locking mechanism using semaphores and MUTEXes. This is a good way to code any real-time threads that require data to be shared among them without corruption of it. The idea is to allow only one thread to work on the shared memory at a time. It can be done by implementing a type of counter, which, when zero, makes a thread wait., and when one or more, makes the thread execute on the shared memory. A MUTEX can be created globally and locked by threads in the critical sections where they update or read global data. This ensures that a thread is not interrupted by another thread during critical update and/or read. In the function, it is to be ensured that update and read are safe from one another. The function `mutex_init()` can be used to initialize a MUTEX in the main thread. The corresponding threads can then use `mutex_lock()` and `mutex_unlock()` functions around the critical sections to make them safe. The drawbacks of this implementation method include unbounded priority inversion or unbounded blocking, which can generally be overcome/fixed with proper coding techniques.

Impact:

Problem with Shared Functions

If a shared function updates global data, it can be a problem since two or more threads accessing it can corrupt the data. In such a condition, it is extremely important to make the thread safe such that it is not interfered with by the other thread, while updating common global data.

Thread-Safe Functions

Thread-safe functions are those that access shared data in a safe manner. This implies that, multiple calls to a thread-safe function, at the same time, will not have any adverse effect on the variables or components handled by the function.

Re-entrant Functions

Re-entrant functions are those that can be interrupted or pre-empted, then called again in a safe manner, without finishing execution of the original call.

Hence, essentially, if a shared function exists such that it can be called by more than one thread and if those threads are concurrently active, the function must provide re-entrant support so that it is thread-safe.

Implementation using MUTEX and RT-Linux Pthreads

This is a design and implementation of a thread-safe application that modifies contents of a structure with timestamps and reads the changes without corrupting global data.

A code using MUTEX and RT-Linux Pthread APIs was written to generate data and timestamp in one thread and print it out safely in another. It is built using a makefile and executed to obtain the output as shown in the following screenshot.

```
-----
Updating the values. Please wait.
Value of X = 2.000000
Value of Y = 2.000000
Value of Z = 2.000000
Value of acceleration = 2.000000
Value of yaw = 2.000000
Value of pitch = 2.000000
Value of roll = 2.000000
Timestamp = 951113069
Reading the updated values
Value of X = 2.000000
Value of Y = 2.000000
Value of Z = 2.000000
Value of acceleration = 2.000000
Value of yaw = 2.000000
Value of pitch = 2.000000
Value of roll = 2.000000
Timestamp = 951113069
-----
Updating the values. Please wait.
Value of X = 5.000000
Value of Y = 5.000000
Value of Z = 5.000000
Value of acceleration = 5.000000
Value of yaw = 5.000000
Value of pitch = 5.000000
Value of roll = 5.000000
Timestamp = 951611761
Reading the updated values
Value of X = 5.000000
Value of Y = 5.000000
Value of Z = 5.000000
Value of acceleration = 5.000000
Value of yaw = 5.000000
Value of pitch = 5.000000
Value of roll = 5.000000
Timestamp = 951611761
-----
```

The code does a thread-safe update of a complex state with a timestamp. It includes two threads - one updates values of all parameters and timestamp contained in a global structure that

includes a double precision attitude state, while the other reads the time-stamped state from the same structure and prints out the entire data without data corruption. Both the threads use a global MUTEX while updating and reading so that the data is never corrupted and the threads remain safe.

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

ANSWER

Root cause of deadlock:

Deadlock in a system occurs when thread 1 has acquired resource A and is waiting to get access to resource B. Likewise, thread 2 has got access to resource B and is waiting to get access for resource A. This kind of circular dependency leads to condition called as deadlock. The cause of deadlock lies in underlying fact that a particular thread holds on to a resource for indefinite amount of time and is not willing to release the resources.

One of the possible fix is to provide a specific time for which a thread can hold on to a resource. If the time period expires and the thread is still holding onto a resource, the thread will be forced to drop the resource so that a thread which is waiting for that resource can grab it and can complete its execution. This is done by using `pthread_mutex_timedlock()` function. However, there is a problem which makes it difficult for this method to operate, what if both the threads release the resource at the same time, this will cause both of the thread to grab the alternate resource which will cause live-lock problem.

To solve this problem, a random back-off timer is used. This will cause one of the thread to release the resource while the other thread is holding onto both of the resources. In the code provided in the URL, we have used the random back-off timer strategy and solved the deadlock problem.

Root cause for priority inversion:

Priority inversion occurs when there are three threads with priority of low, medium and high, out of which low and high priority thread share a MUTEX for a particular critical section, while medium priority thread does not share any memory with any of the two threads

What happens is, when lower priority thread is running in the critical section, higher priority thread cannot preempt lower priority thread as it is protected by MUTEX.

Now, if a middle priority thread arrives, it will preempt lower priority thread which in turn blocks the execution of higher priority thread, this condition is called as priority inversion.

One way to fix this priority inversion is by use of priority inheritance protocol wherein the priority of lower priority thread is amplified to that of higher priority, hence middle priority thread cannot preempt it.

Another way is to use priority ceiling protocol, wherein the priority of lowest priority thread is amplified to a very higher bound value determined by the programmer. This technique requires the programmer to know the priority of all threads which are running in the given system.

The problem was fixed in the example code provided in the URL by temporarily boosting the priority of lower priority thread to higher priority using priority inheritance protocol.

For the unbounded inversion, is there a real fix in Linux?

Nope, we cannot fix unbounded priority inversion without help of real time patch in linux because in linux there is no way that the priority of system calls can be changed.

RT_PREEMPT Patch, but would this really help?

Given that priority inheritance protocol cannot be directly implemented in linux, there is a limitation that the priority of any system call or any interrupt cannot be changed. This problem can be solved by using the RT_PREEMPT patch which allows the user to change the priority of any system call. Thus when there is any kind of priority inversion, the priority of lower priority task can be boosted to prevent priority inversion

That being said, the RT_PREEMPT patch does not guarantee that priority inversion will never occur. It however makes sure that any priority inversion that occurs thereafter will be bounded priority inversion and not an unbounded one.

Thus, we can directly include the time of bounded priority inversion in our calculation of our response time and make sure that our system does not gets trapped in priority inversion problem.

The Linux PREEMPT_RT Patch – A Solution for Priority Inversion?

For unbounded inversion in Linux, the priority inheritance protocol can be used. The basic priority inheritance protocol bounds the blocking period to a maximum of 'm' times if there are 'm' number of semaphores being used. With the priority ceiling protocol, the blocking is restricted to only once, and this also prevents deadlocks. The PI-Futex already present in Linux kernel can be

used to implement priority inheritance. However, inversion might still be present due to in-kernel spinlocks, MUTEXes or semaphores.

The Linux PREEMPT_RT real-time patch is a patch that exists to fix the issues of Linux being able to meet only soft real-time deadlines and those that arise from using PI-Futex. It converts Linux into a fully pre-emptible kernel and attempts to address the issue of priority inversion using priority inheritance. Priority Inheritance is when a high priority thread is blocked and made to wait because a lower priority thread is using the same shared memory, the low priority thread will inherit the priority of the higher priority thread's priority for the duration of its use of the shared memory. The low priority thread can thus no longer be pre-empted by medium priority threads. This method addresses the unbounded priority inversion problem in the sense that the high priority thread can no longer be stuck and made to wait behind infinite medium priority threads., it is blocked only for the duration during which the low priority thread is using the shared memory. The in-kernel locks can be pre-empted by using RTMUTEXes instead of the traditional locks. Even the critical sections guarded by spinlocks can be pre-empted, unless they are implemented using raw_spinlock_t. The interrupt handlers are also pre-emptible and they are treated as a kernel thread. The old timer API was changed to obtain high resolution kernel timers and timeouts. Thus with the PREEMPT_RT patch, Linux can be used to implement a hard real time system. The system level locks, critical sections and interrupts can be pre-empted, they can be used for priority inheritance to avoid priority inversion.

The limitation of this method is that the low priority thread gets high priority for much of its execution duration, which is undesired. If the high priority thread is using the shared memory of an even higher priority thread, the low priority thread could even be pushed all the way up to that higher priority. In a development point of view, an implementation can be looked at, where the low priority thread would never execute its critical section if pre-empting the higher priority thread. Such a system would limit blocking time for high priority tasks and would be extremely advantageous, but would be harder to implement. In such cases, Priority Inheritance becomes a solution. Agreed that it does not address the entire issue of priority inversion and that it does have limitations, it is to be noted that it does address and solve the problem of unbounded priority inversion.

The question, whether to switch to an RTOS because of existing priority inversion problems in Linux is interesting. While an RTOS might address the priority inversion issue a lot better than Linux (even with the pre-empt patch), it does not come with the huge community and the great volume of code that has gone into Linux. With the PREEMPT_RT patch, Linux can be used for hard real-time applications too, apart from soft real-time applications. This being said, for projects dealt by companies where the application is extremely hard real-time and much of the code should be written from scratch, an RTOS will be a better choice since it is more specifically tailored to real-time applications than Linux. On the other hand, Linux provides a multitude of non-embedded system tools that might be extremely useful for some applications. In such cases, Linux

implementation with the pre-empt patch and Priority Inheritance to solve the inversion issue, is a better option.

Though RT_PREEMPT_PATCH does solve the problem of priority inversion, given the huge memory footprint of linux kernel, there may be area or space in linux, where there might be some possibility of unbounded priority inversion. If such priority inversion occurs, it will cause hard real time system to miss its deadline and may result in system failure.

Hence, as we know that linux is not hard real time operating system, it may be difficult for linux to emulate the functionality of hard real time operating system even if certain real time patches are applied.

Thus, to get functionally correct output within given time period, we must rely on RTOS and not use linux for hard real time services.

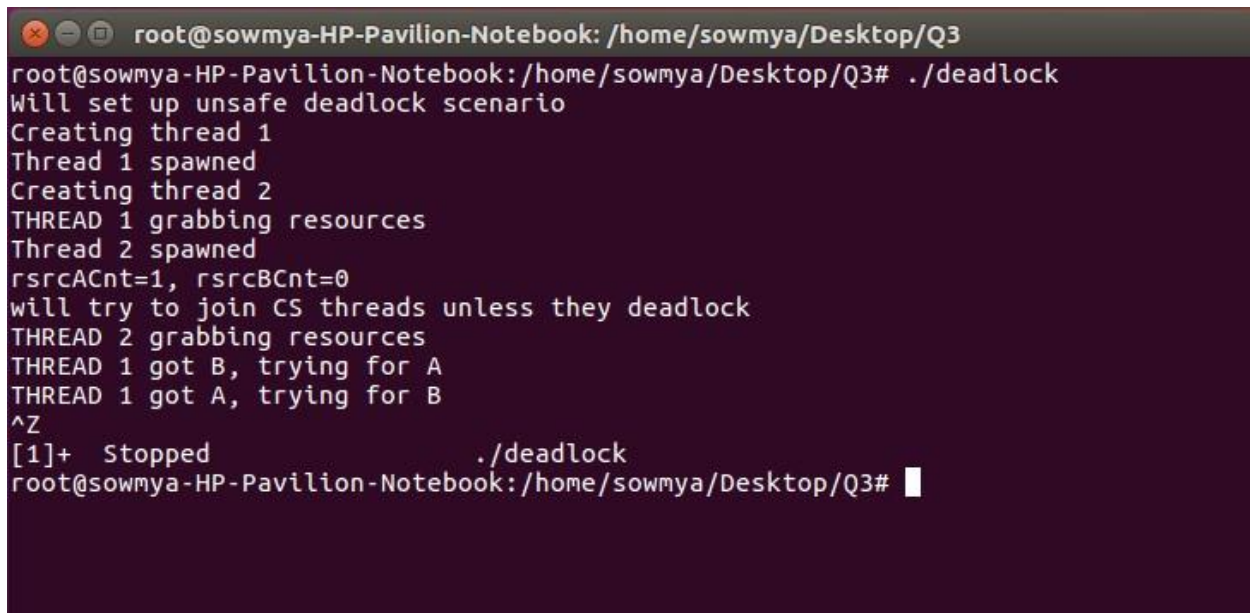
Implementation and correction of code

The given code is implemented and the occurrence of deadlock and unbounded priority inversion are analyzed. The code is then modified to avoid the issues.

The implementations are as follows.

deadlock.c

The given code for deadlock has two threads. The code ends in deadlock.

A terminal window with a dark background and light-colored text. The window title is 'root@sowmya-HP-Pavilion-Notebook: /home/sowmya/Desktop/Q3'. The user has entered the command './deadlock'. The output shows the program setting up an unsafe deadlock scenario, creating two threads, and then entering a state where both threads are holding one resource and waiting for the other, resulting in a deadlock. The user presses Ctrl+C (^Z) to stop the program, and the prompt returns.

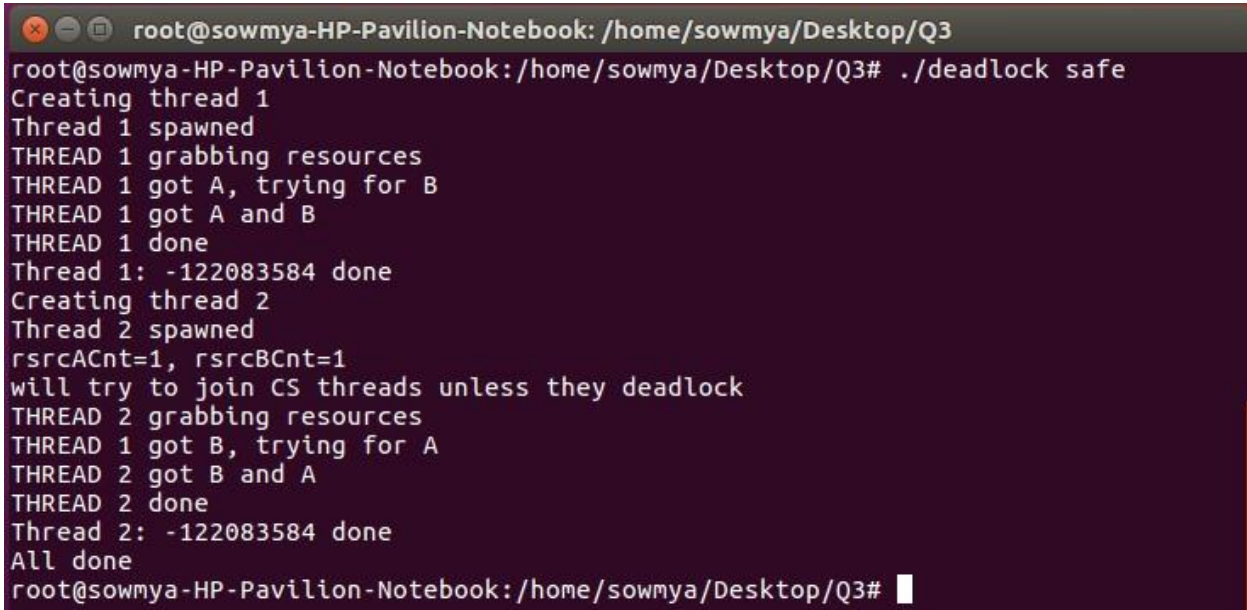
```
root@sowmya-HP-Pavilion-Notebook: /home/sowmya/Desktop/Q3
root@sowmya-HP-Pavilion-Notebook: /home/sowmya/Desktop/Q3# ./deadlock
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
THREAD 1 grabbing resources
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 got B, trying for A
THREAD 1 got A, trying for B
^Z
[1]+  Stopped                  ./deadlock
root@sowmya-HP-Pavilion-Notebook: /home/sowmya/Desktop/Q3#
```

The main thread creates thread-1 and thread-2, waits for thread-1 to end, then waits for thread2 to end. But thread-1 first grabs resource-A and goes into sleep state for 1 second during which thread-2 runs and grabs resource-B and then goes into sleep state for 1 second. Thread-1 then comes out of the sleep state and tries to grab resource-B but is not able to get it as thread-2 has

locked it. In a similar way thread-2 tries to grab resource-A which is locked by thread-1. This leads to deadlock condition.

Now, the code provides the provision of passing safe and race as arguments and we observe its functioning as follows.

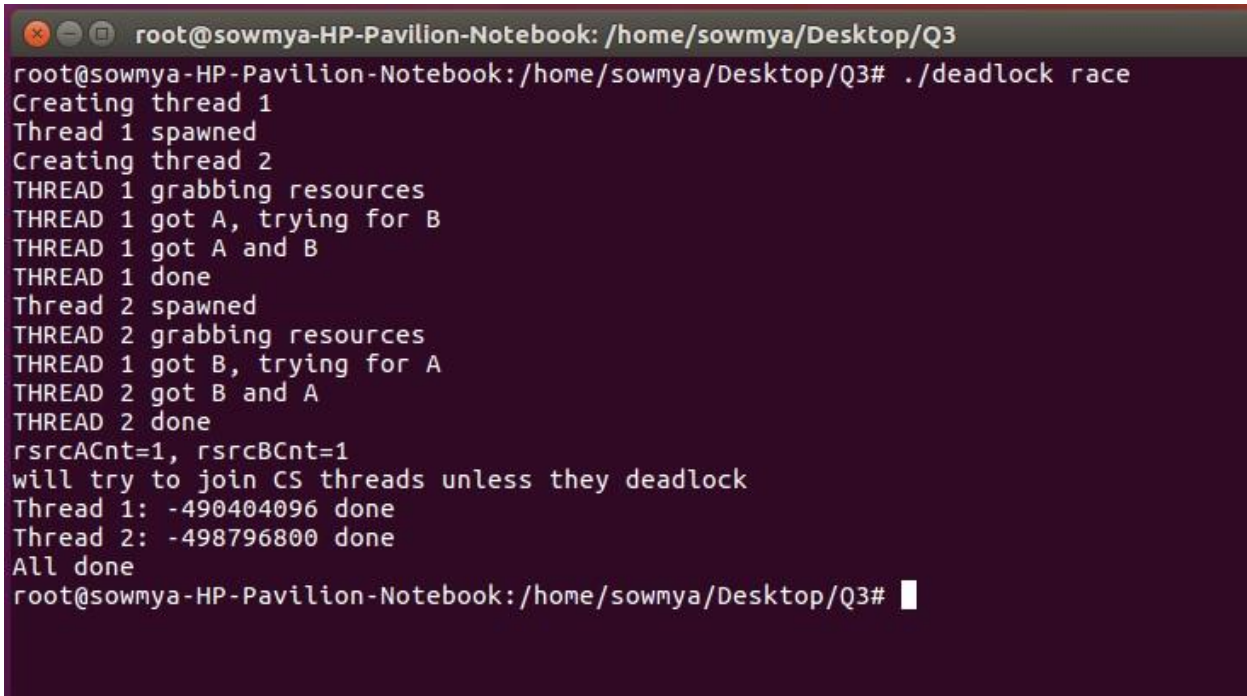
deadlock_safe.c



```
root@sowmya-HP-Pavilion-Notebook: /home/sowmya/Desktop/Q3
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Desktop/Q3# ./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: -122083584 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: -122083584 done
All done
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Desktop/Q3#
```

When the safe attribute is passed, thread-1 is first generated and the function waits till the completion of its execution, after which thread-2 is created and the same wait process is repeated. This way, the two threads never run simultaneously and hence, deadlock is avoided.

deadlock_race.c

A terminal window with a dark purple background and white text. The window title is 'root@sowmya-HP-Pavilion-Notebook: /home/sowmya/Desktop/Q3'. The command './deadlock race' has been executed. The output shows the creation and execution of two threads. Thread 1 acquires resource A first, then B. Thread 2 acquires resource B first, then A. Both threads complete their execution successfully. The program then prints resource counts and thread IDs, indicating no deadlock occurred.

```
root@sowmya-HP-Pavilion-Notebook: /home/sowmya/Desktop/Q3
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Desktop/Q3# ./deadlock race
Creating thread 1
Thread 1 spawned
Creating thread 2
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 2 spawned
THREAD 2 grabbing resources
THREAD 1 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
rsrcACnt=1, rsrcBCnt=1
will try to join CS threads unless they deadlock
Thread 1: -490404096 done
Thread 2: -498796800 done
All done
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Desktop/Q3#
```

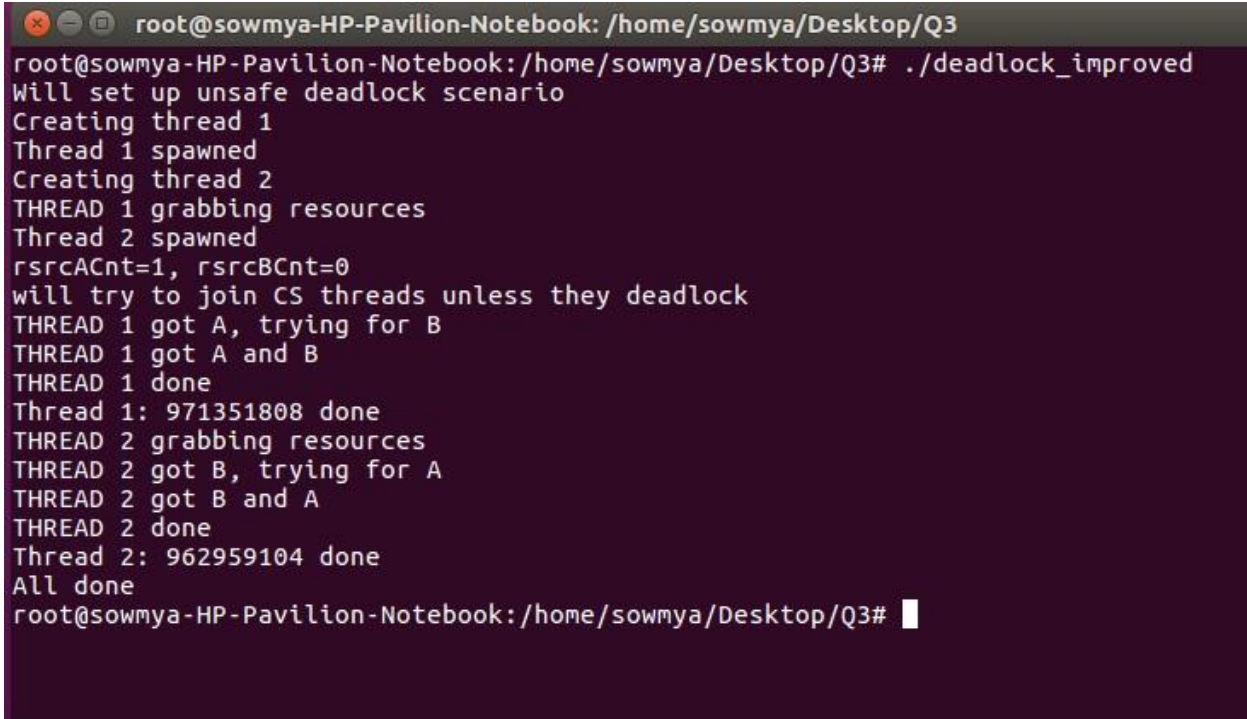
When the race attribute is passed, both the threads are generated one after the other, the 'no wait' flag becomes 1 and hence the threads do not block the resources. As resources are not blocked, nothing can be said in certain about the execution. As seen from the above screenshot, we observe that there is no deadlock as no thread is waiting on the resource. Under race condition, nothing can be certainly said about the occurrence of deadlocks.

deadlock_timeout.c

```
root@sowmya-HP-Pavilion-Notebook: /home/sowmya/Desktop/Q3
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Desktop/Q3# ./deadlock_timeout
Will set up unsafe deadlock scenario
Creating thread 1
Creating thread 2
Thread 1 started
THREAD 1 grabbing resource A @ 1520469590 sec and 859418154 nsec
Thread 2 started
THREAD 2 grabbing resource B @ 1520469590 sec and 859509955 nsec
Thread 2 GOT B
rsrcACnt=0, rsrcBCnt=1
will try to join both CS threads unless they deadlock
Thread 1 GOT A
rsrcACnt=1, rsrcBCnt=1
THREAD 2 got B, trying for A @ 1520469591 sec and 860183719 nsec
THREAD 1 got A, trying for B @ 1520469591 sec and 860221581 nsec
Thread 2 TIMEOUT ERROR
Thread 1 GOT B @ 1520469593 sec and 860832446 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
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Desktop/Q3#
```

In `deadlock_timeout.c`, the two threads are created first. Thread-1 starts and takes resource-A; one nanosecond later, thread-2 starts and take resource-B. The time is calculated and provided in this code. After executing for some nanoseconds, thread-2 tries getting resource-A; the same happens with thread-1 trying to get resource-B. Unable to do so because resource-A is still locked by thread-1 at the time when thread-2 is trying to get it, thread-2 produces a timeout error and stops execution. At and after this point, the resource-B held by thread-2 is free. When thread-1 thus tries accessing it after some nanoseconds after timeout of thread-2, it is able to get resourceB. With both resource-A and resource-B with it, thread-1 thus completes successful execution. This is a situation where one thread goes into timeout state and the other, thus, is able to complete its execution.

We now implement the Random Back Off method for preventing deadlocks. Here we delay one of the threads by some random time so that, before one thread requests for a common resource, the other thread has completed. The implementation has been described below: **deadlock_improved.c**



```
root@sowmya-HP-Pavilion-Notebook: /home/sowmya/Desktop/Q3
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Desktop/Q3# ./deadlock_improved
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
THREAD 1 grabbing resources
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: 971351808 done
THREAD 2 grabbing resources
THREAD 2 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
Thread 2: 962959104 done
All done
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Desktop/Q3#
```

Here, the code is modified such that, when thread-1 grabs resource-A and goes to sleep, thread2 is made to back-off for some random time which is more than the time for which thread-1 sleeps. This ensures that there is enough time for thread-1 to acquire both the resources and complete its execution and the execution of thread-2 will then begin, which will also get both the resources and be able to complete its execution as well.

To understand unbounded priority inversion, we execute the pthread3 and pthread3ok codes provided to us.

pthread3.c

```
root@sowmya-HP-Pavilion-Notebook: /home/sowmya/Desktop/Q3# ./pthread3
Usage: pthread interfere-seconds
root@sowmya-HP-Pavilion-Notebook: /home/sowmya/Desktop/Q3# ./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
Start services thread spawned
will join service threads
Creating thread 3
Low prio 3 thread spawned at 0 sec, 327886 nsec
Creating thread 2
Middle prio 2 thread spawned at 1 sec, 1052660 nsec
Creating thread 1, CScnt=1
**** 2 idle NO SEM stopping at 1 sec, 1056952 nsec
High prio 1 thread spawned at 1 sec, 1202211 nsec
**** 3 idle stopping at 2 sec, 881648 nsec
LOW PRIO done
MID PRIO done
**** 1 idle stopping at 4 sec, 1227336 nsec
HIGH PRIO done
START SERVICE done
All done
root@sowmya-HP-Pavilion-Notebook: /home/sowmya/Desktop/Q3#
```

In pthread3.c, there are three services allotted low, medium and high priorities. The interference time is 1 second. Since Services 1 and 3 share the same resources, the low and high priority services have to use MUTEX to perform their functions while the medium priority function can execute without it. The low priority service runs and locks the resource. The medium priority service, which does not require MUTEX, attempts to pre-empt the low priority service. The high priority service further pre-empts the medium priority service. However, since the resource is blocked by the low priority service, the execution of high priority service cannot be completed. It is thus noticed that, the low priority service finishes first and high priority service finishes last. This is the depiction of Priority Inversion. Even though the high priority service pre-empts the low and medium priority services, it cannot execute due to the MUTEX lock on the required service and also has to wait till the medium priority service executes. In a hard real-time system, this is harmful.

pthread3ok.c

```
root@sowmya-HP-Pavilion-Notebook: /home/sowmya/Desktop/Q3
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Desktop/Q3# ./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
Start services thread spawned
will join service threads
Creating thread 1
High prio 1 thread spawned at 0 sec, 356792 nsec
**** 1 idle stopping at 0 sec, 373141 nsec
Creating thread 2
Middle prio 2 thread spawned at 0 sec, 524365 nsec
Creating thread 3
**** 2 idle stopping at 0 sec, 582905 nsec
Low prio 3 thread spawned at 0 sec, 669272 nsec
**** 3 idle stopping at 0 sec, 682501 nsec
LOW PRIO done
MID PRIO done
HIGH PRIO done
START SERVICE done
All done
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Desktop/Q3#
```

In pthread3ok.c, there are three services present, no MUTEXes and subsequently, no resource blocking. The high priority task finishes first followed by the medium and then the low. There is absence of priority inversion in this case. Such an implementation is dangerous as the common resource/data might get corrupted while being read from and written to simultaneously by two threads. A possible solution would be to implement Priority Inheritance and Priority Ceiling protocols. When hard real-time applications are involved, they would execute properly in the order of priority and save data from being corrupted.

4) [15 points] Review [heap_mq.c](#) and [posix_mq.c](#). First, re-write the VxWorks code so that it uses RT-Linux Pthreads (FIFO) instead of VxWorks tasks, and then write a brief paragraph describing how these two message queue applications are similar and how they are different. You may find the following [Linux POSIX demo code](#) useful, but make sure you not only read and port the code, but that you build it, load it, and execute it to make sure you understand how both applications work and prove that you got the POSIX message queue features working in Linux on your Altera DE1-SoC or Jetson. Message queues are often suggested as a way to avoid MUTEX priority inversion. Would you agree that this really circumvents the problem of unbounded inversion? If so why, if not, why not?

ANSWER

Certain real time operating system have priority inheritance message queue which means that any middle priority thread cannot preempt low priority thread. Message queue are implemented to synchronize passing of message from sender to receiver thread without any priority inversion. Thus, the mqsend() and mqreceive function successfully allow us to avoid any priority inversion

Thus using message queues, the message can be passed from sender to receiver thread. If there occurs a situation wherein the queue is empty, the receiving thread has to wait until the queue becomes full. Similarly, the sending thread has to wait sometime until the receiving thread does not receive message from message queue that is full.

The messages have priority assigned to them, thus the highest priority message will be de-queued first before lower priority message thus solving the problem of global memory sharing.

SIMILARITIES

Both functions:

- Give the receiver a higher priority than the sender.
- Ensure a safe way for the threads or processes to communicate with each other.
- Use tasks to run the sender and receiver functions.
- Use the POSIX queue for storing the messages with the function calls mq_open, mq_receive and mq_send.

DIFFERENCES

The first major difference between the two implementations is that heap_mq.c creates and opens the message queue before running the sender and receiver functions. The posix_mq.c code opens and creates the message queue in the receiver function and it is again opened in the sender function.

The other major difference is that heap_mq.c uses dynamically allocated memory to be used as the buffer, while posix_mq.c uses stack memory.

In message queue, the message transferred is the actual data and in the heap queue, the data transferred are actually pointers to messages. The pointers are set to point to a buffer allocated by the sender, and the pointer received is used to access and process the buffer. Hence in the heap queue, the queue is a buffer of pointers.

Heap_mq.c also implements the sender and receiver functions as infinite loops that must be closed by deleting the tasks, while posix_mq.c only runs through the sending and receiving functions once.

In message queue, it is important to ensure that the read and write to the buffer is atomic using some semaphore or mutex. Such a restriction is not necessary in case of heap queue.

For message queue the size of the message queue always remains constant which is determined when the queue is opened, but in case of the heap queue, the sender allocates the buffer and the receiver de-allocates to avoid exhaustion of the associated buffer heap.

Similarities	Differences
Both of them uses task spawn function to spawn threads	The heap_mq function relies on dynamic memory allocation technique whereas Posix_mq does not
Both of them uses sender and receiver message queues	In heap_mq file, the image buffer is used as the message to be operated on , while in posix_mq it is the canned message which is operated
The function of both program is the same.	In heap_mq, we have used the shutdown operation function towards the end of source file, while in case of posix_mq we haven't used any kind of shutdown operation.
Both the codes have used error reporting mechanism, wherein if the queue is failed to create, a corresponding message is generated using perror function.	In heap_mq, we have used memcpy function many times which may increase the size of executable, while in case of posix_mq, we haven't used any.

Posix_mq

Screenshot:

```
root@vikrant-VirtualBox:/home/vikrant/exercise3/Question4# ./posixmq
recive:msg This is a test, and only a test, in the event of real emergency, you would be instructed.... received with priority =30, length =93
send: message successfully sent
```

Explanantion:

Posix_mq.c

- 1) The code has implementation of two queues, one at sender side and one at receiver side.
 - 2) For sender side queue, we use the function `mq_open` (name, flag, permission, attribute) to create a new message queue which is identified by its name `SNDRCV_MQ`. The second attribute named flag controls operation of call. The flag here is `O_RDWR` which instructs to use the queue for sender as well as receiver. The third argument is the permission which in this case is not required hence zero. The fourth parameter is attribute which tells the maximum number of message and maximum size of message that a queue can allow
 - 3) If there is any error while creating a queue, we print the error message using `perror`.
 - 4) We then send the message pointed by message pointer which is `cannedmsg` array[] created above to message queue referred by message queue description which is the first argument named as `mymq`. The third argument gives the length of message pointed by message pointer. The fourth argument is non negative integer that specifies priority of message placed on queue in order of decreasing priority, with newer message being placed after older message. In this case the message priority is set to 30
 - 5) If the message is sent successfully, a corresponding message appears on the screen or else we get an error.
-
- 1) On receiver side, we define another queue named `SNDRCV_MQ` queue which can be used for sender and receiver and attributes specified by last parameter.
 - 2) If there is any error, any error message is thrown
 - 3) Then, the `mq_receive`(message descriptor, message pointer, length, priority) function then removes the message pointed by first argument and places that message in the queue pointed by second parameter which is buffer in this case, the maximum size of message is defined by third parameter while the priority is defined according to fourth parameter.
 - 4) If this is done successfully, we get the message on screen that tells us which message is received and its priority along with its length.
-
- 1) The `mq_demo` function sets the common attributes such as maximum message and maximum message size. It also sets flag required for its operation.

- 2) Then it uses `taskspawn()` function to create thread 1, the first parameter gives it name which is receiver , the second parameter defines priority which is 90, fourth defines the stack size,the fifth parameter defines the function to it must jump when it is created and the argument parameter is kept zero.
- 3) After receiver thread is created, we then create the sender thread that calls sender function with priority of 100, stack size 400. It gives error if corresponding task is not successfully created or gives message of success if thread is spawned successfully.

Heap_mq

Screenshot:

```
buffer =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Reading 8 bytes

Receiver Thread Created Sucessfully!

Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes

Receive: ptr msg 0x0x7f3c200008c0 received with priority = 30, length = 12, id = 999

Contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

Heap space memory freed

Reading 8 bytes

Send: message ptr 0x0x7f3c200008c0 successfully sent

Sender Thread Created Sucessfully!

Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes

Receive: ptr msg 0x0x7f3c200008c0 received with priority = 30, length = 12, id = 999

Contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

Heap space memory freed

Reading 8 bytes

Send: message ptr 0x0x7f3c200008c0 successfully sent
```

Explanation:

- 1) In receiver function, a buffer is created and buffer pointer is initialized
- 2) `mq_receive()` function is used, which takes the message in queue `mymq` and places it in queue pointed by buffer pointer, the length is given in third parameter and in fourth parameter, the priority of message is defined.
- 3) If the `mq_receive` function gives error, it is displayed accordingly else, the message from buffer is copied to `buffptr`. Also, the content is copied to specified thread id.
- 4) After copying is done the corresponding memory is freed in order to avoid fragmentation.

- 1) On sender side, the malloc function call is used for buffptr to reserve set of memory location in heap.
- 2) Using string copy function, the string is copied from imagebuff to buffptr.
- 3) Then the content is copied using memcpy function from buffptr to buffer.
- 4) Using the function mq_send(), the message is sent from buffer to mymq, length is given in third parameter, and priority is given in fourth parameter. If there is any error, then it gives an error message or else the message is sent successfully.

- 1) In the heap_mq function, the imagebuffer is initialized using for loop
- 2) Then the attributes are initialized, the maximum number of message are defined, also the message size is defined. The flag is initialized as zero. The mq_open() function creates a new queue SNDRCV_MQ which used for sending as well as receiving and set with attributes defined above.
- 3) The tasks are spawned with help of task spawn function. The sender task given priority of 100 and receiver task given priority of 90.
- 4) A shutdown function is written which deletes the message queue and deletes sender and receiver task.

Message Queues – The solution to Unbounded MUTEX Priority Inversion?

Priority inversion is an issue when MUTEXes and semaphores are used, which do resource blocking in order to modify global data. Removing MUTEXes and semaphores is not feasible and advisable since global data is invariably required for communication between threads. An alternative to this would be to use Message Queues – they prevent the problem of unbounded priority inversion and provide a method for establishment of communication between threads and processes without locking resources. However, normal queues cannot always ensure the absence of priority inversion. POSIX message queues, to ensure that priority inversion does not take place, have a feature of queueing and servicing messages according to priority in a timely fashion. Message queues ensure that dequeue of messages takes place such that the one with the highest priority gets dequeued first. Thus, priority can be given to the messages in a way that priority inversion does not occur. The higher priority task, waiting on the lower priority task, waits for a deterministic amount of time, thus avoiding unbounded priority inversion. The enqueue and dequeue operations are thread-safe.

- 5) [35 points] Watchdog timers, timeouts and timer services – First, read this overview of the [Linux Watchdog Daemon](#) and describe how it might be used if software caused an indefinite deadlock. Next, to explore timeouts, use your code from #2 and create a thread that waits on a MUTEX semaphore for up to 10 seconds and then 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.

ANSWER

A watchdog in computer is basically a hardware, which supervises the system behavior and if it fails to perform, a reset operation is performed to recover.

It is often the last resort to maintain the system and it doesn't stop hardware fault from breaking a system, nor it is good for any kind of software issues. But as a Whole it improves the overall reliability/availability of system.

There are two parts to watchdog namely:

- 1) The actual hardware timer and kernel module that performs forced reset
- 2) A user background daemon that refreshes the timer and provides more range of recovery option.

Both of them function independently and provide maximum protection.

The watch dog module:

It is a simple timer that is set to a reasonable timeout, and periodically refreshed by software. If for any reason the software stops refreshing the hardware, then the timer times out and performs hardware reset of the whole system. In case of Linux OS, there is standard interface to watchdog hardware provided as /dev/watchdog. This driver is not default and may have to be manually configured. This is done by adding module name to /etc/modules.

The watchdog Daemon

To operate the watchdog device, there is a background task known as daemon that refresh the system periodically. Sometimes, the system goes into some unused state without terminating the daemon which is running in the background and hence the daemon has to be configured by running set of tests on the system to know that the system is functioning properly. On failing such tests the daemon reboots the machine in orderly fashion, keeps a log to know the reason. In case the daemon fails to reboot, there is hardware reset which will reset it.

Description of how the Linux WD timer can help with recovery from a total loss of software sanity (E.g. system deadlock)

When a system with two process is trapped in deadlock, it implies that process 1 is holding resource 1 and waiting to get resource 2, while process 2 is holding resource 2 and waiting for

process 1 to release resource 1. This kind of looped condition where one of the process depends on another to release resource is often termed as deadlock and it will cause the system to get stuck into particular state. A watchdog timer can be effective in such cases, the watchdog timer is basically a hardware timer that starts counting up to a pre-specified count and when the value is reached it resets the whole system. There is usually a background task known as daemon which periodically refreshes the hardware timer.

So when the system is in deadlock condition, the watchdog timer starts, if the system is still under deadlock before the watchdog timer reaches its limit then the daemon does not refreshes the timer. Thus when the final count is reached, the system is rebooted. By rebooting the system, both the process which are in deadlock condition are terminated and this causes both of them to release the resource. Once the resources are released, the process can be scheduled in such a way that they can acquire both the resources and then they no longer have to wait for another process which is holding the resource. This is how watchdog daemon can help in solving deadlock condition.

Modification done in problem 2 code to implement timed mutex lock:

```
void *getvalues(void *arg){
int i=-1;
while(i!=0){
tensecond.tv_sec=10;
i=pthread_mutex_timedlock(&mutex,&tensecond);    // Mutex timed lock
printf("No new data available at %ld\n",attitude.timestamp.tv_sec);
}
```

Explanation:

The code is modified by using a function called as a `mutex_timedlock()` function which behaves as follows:

Initially, we specify the time duration for which the mutex lock cannot be acquired. After the specified time out expires, the reader thread can then acquire mutex and read the updated values. The writer thread is put to sleep for more than 10 seconds in screenshot 2 to give the effect that it has acquired the mutex lock for more than 10 seconds and hence the terminal shows that data is not available for time period until the sleep time gets expired.

The above screenshot shows the usage of `timedlock` mutex wherein the second parameter tells us the timeout which is 10 secs in this case. SO, the reader thread cannot get the mutex lock until a timeout of 10 seconds expire. After the time out expires, the timed mutex function returns zero, which causes the basic block to exit the while loop and continue execution in its critical section.

Screenshot of problem5 using timed mutex lock:

When we do not allow reader thread to acquire mutex lock within given time of 10seconds.

NOTE: we are using sleep function within writer thread to simulate the effect that the writer thread is acquiring mutex lock for more than 10 seconds.

```
No new data available at 1520534970
No new data available at 1520534970
No new data available at 1520534970
Reading the updated values
Value of X = 5.000000
Value of Y = 5.000000
Value of Z = 5.000000
Value of acceleration = 5.000000
Value of yaw = 5.000000
Value of pitch = 5.000000
Value of roll = 5.000000
Timestamp = 890840777
```

When we don't put sleep function and allow reader thread to acquire the mutex lock and hence enters critical section

```
root@vikrant-VirtualBox:/home/vikrant/exercise3/Question5# ./problem5
-----
Updating the values. Please wait.
Value of X = 2.000000
Value of Y = 2.000000
Value of Z = 2.000000
Value of acceleration = 2.000000
Value of yaw = 2.000000
Value of pitch = 2.000000
Value of roll = 2.000000
Timestamp = 354393816
No new data available at 1520533368
Reading the updated values
Value of X = 2.000000
Value of Y = 2.000000
Value of Z = 2.000000
Value of acceleration = 2.000000
Value of yaw = 2.000000
Value of pitch = 2.000000
Value of roll = 2.000000
Timestamp = 354393816
-----
-----
Updating the values. Please wait.
Value of X = 5.000000
Value of Y = 5.000000
Value of Z = 5.000000
Value of acceleration = 5.000000
Value of yaw = 5.000000
Value of pitch = 5.000000
Value of roll = 5.000000
Timestamp = 355289686
No new data available at 1520533368
Reading the updated values
Value of X = 5.000000
Value of Y = 5.000000
Value of Z = 5.000000
Value of acceleration = 5.000000
Value of yaw = 5.000000
Value of pitch = 5.000000
Value of roll = 5.000000
Timestamp = 355289686
-----
root@vikrant-VirtualBox:/home/vikrant/exercise3/Question5#
```

REFERENCES

1. ["Priority Inheritance Protocols: An Approach to Real-Time Synchronization" by Lui Sha, Ragunathan Rajkumar, John P. Lehoczky](#)
2. <http://mercury.pr.erau.edu/~siewerts/cec450/documents/Papers/soc-5.pdf>Scheduling
3. "Priority Inversion in the Kernel" - <https://lwn.net/Articles/178253/>
4. "PI=futex: - VI" <https://lwn.net/Articles/177111/>
5. Real-Time Embedded Components and Systems – Dr. Sam Siewart
6. Linux Man Pages

Grading Rubric

[10 points] Priority inheritance paper review and unbounded priority inversion issues:

[4 points] Three Priority inheritance key points made in paper _____ [3

points] Why the Linux position makes sense or not _____ [3 points]

Reasoning on whether FUTEX fixes _____ [25 points] Thread

safety with global data and issues for real-time services:

[5 pts] Description of methods _____

[5 pts] Impact of each to real-time services _____

[15 pts] Correct shared state update code _____

[15 points] Example synchronization code using POSIX threads:

[5 pts] Demonstration and description of deadlock with threads

[5 pts] Demonstration and description of priority inversion with threads

[5 pts] Description of RT_PREEMPT_PATCH and assessment of whether Linux can be made real-time safe _____ [15 points]

Example synchronization code using POSIX message queues and threads:

[10 pts] Demonstration of message queues on Altera DE1-SoC adapted from examples

[5 pts] Description of how message queues would or would not solve issues associated with global memory sharing

[15 points] Watchdog timers (for the system) and timeouts (for API calls):

[10 pts] Description of how the Linux WD timer can help with recovery from a total loss of software sanity (E.g. system deadlock)

[15 pts] Adaptation of code from #2 MUTEX sharing to handle timeouts for shared state

Overall, provide a well-documented professional report of your findings, output, and tests so that it is easy for a colleague (or instructor) to understand what you've done. Include any C/C++ source code you write (or modify) and Makefiles needed to build your code. I will look at your report first, so it must be well written and clearly address each problem providing clear and concise responses to receive credit.

Note: Linux manual pages can be found for all system calls (e.g. fork()) on the web at <http://linux.die.net/man/> - e.g. <http://linux.die.net/man/2/fork>

In this class, you'll be expected to consult the Linux manual pages and to do some reading and research on your own, so practice this in this first lab and try to answer as many of your own questions as possible, but do come to office hours and ask for help if you get stuck.

Upload all code and your report completed using MS Word or as a PDF to D2L and include all source code (ideally example output should be integrated into the report directly, but if not, clearly label in the report and by filename if test and example output is not pasted directly into the report). ***Your code must include a Makefile so the TAs can build your solution on Ubuntu VB-Linux, a Jetson or a Altera DE1-SoC. Please zip or tar.gz your solution with your first and last name embedded in the directory name.***