

Introduction

The purpose of this exercise is to dive deeper into synchronization problems and explore techniques which attempt to solve these issues. We used a **Jetson TK1 board** for the code that we wrote in this exercise and all results obtained should be reproducible with the appropriate reference material and included scripts.

Problem 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 unbounded priority inversion as described in the paper? If not, what is different about It?

Solution:

“Priority Inheritance Protocols: An Approach to Real-Time Synchronization”:

The purpose of this paper is to investigate the class of priority inheritance protocols - i.e. *basic priority inheritance protocol* and *priority ceiling protocol* - in order to solve the uncontrolled priority inversion problem. The uncontrolled priority inversion problem occurs when a higher priority job is blocked by a lower priority job for an indefinite period of time due to the use of synchronization primitives such as semaphores, monitors or the Ada rendezvous. This can often result in missed deadlines and system failures if the period of blocking lasts too long. Ultimately, they were able to conclude that the *priority ceiling protocol* reduces the worst case task blocking time to only the duration of the lower priority task meaning that this can prevent the occurrence of deadlocks in the system.

The paper specifically talks about two priority inheritance protocols:

1. Basic Priority Inheritance Protocol: The idea behind the basic priority inheritance protocol is that if a lower priority job blocks a higher priority job, it will execute the critical section of the highest priority level of all the jobs that it blocks. Upon completion

of critical section, it then reverts to its original priority level allowing it to be preempted by other higher level tasks.

Note that the text states that a higher priority job can be clocked by a lower priority job for the following reasons:

- Direct blocking: i.e. a situation in which a higher priority job attempts to lock a locked semaphore. This is necessary to ensure the consistency of shared data.
- Push Through Blocking: i.e a situation in which a medium priority job can be blocked by a low priority job inheriting the priority of a higher job.

Therefore, from the numerous lemmas in section 3.B, they are able to demonstrate that the basic priority inheritance still doesn't address the issue of deadlocks despite being a good alternative to solving the priority inversion problem. This is due to:

- Behavior of Nested Critical Sections: Think of a scenario in which two different priority tasks try to lock a semaphore which the other task has already locked and we have a nested deadlock instance
- "Chain of blocking": This is the case in which a lower priority task enters a critical section and attempts to lock a semaphore which has already been locked and therefore a task would be blocked for the duration of 2 critical sections, one to wait for lock S1 to be released and one for lock S2 to be released.

2. Priority Ceiling Protocol: This protocol is intended to prevent the formation of deadlocks as well as chained blocking. The main idea is to ensure that when a critical section is preempted, the priority of the new critical section to execute is guaranteed to be higher than the inherited priorities of all the preempted critical sections. In order to guarantee this, the idea is to first assign a priority ceiling to each semaphore - equal to the highest priority task that may use this semaphore. A job with higher priority is then allowed to start a new critical section if all semaphores are locked by other job other than J.

However, in this process, it introduces a new type of blocking:

- *Ceiling blocking*. This new type of blocking's worst case scenario is still much better than direct and push-through blocking because a job J can be blocked for at most the duration of one longest subcritical section.

3. Schedulability Analysis: This text builds on the conclusions of Liu and Layland in that it uses their theorem that a set of n periodic tasks will always meet their deadlines if:

$$\frac{C_1}{T_1} + \dots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1)$$

where C_n and T_n are the execution times and periods of the Tasks. Note that the assumptions they made also aligned with what we've discussed in class so far in that they assume all the tasks are:

- All tasks are periodic
- Each job has deterministic execution times and will execute to completion when it is only job in the system
- All periodic tasks are assigned priorities according to Rate Monotonic Policy.

Therefore, building on Rate Monotonic theory, they are able to propose a theorem which states that a set of n periodic tasks using the priority ceiling protocol can be scheduled if the following conditions are satisfied:

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

4. Implementation: Finally, the paper discusses the considerations which should be made the implementation of basic priority inheritance protocol and priority ceiling protocol. Specifically, the basic priority inheritance protocol requires a priority queue of jobs blocked on a semaphore and the system calls the priority inheritance operation. In the case of priority ceiling protocol, there is no longer a ready queue which is needed and this is replaced by a single job queue which is a priority-ordered list of jobs ready to run and be blocked by ceiling protocol. Also, to apply the ceiling protocol monitor, each monitor is assigned a priority ceiling and a job J can enter a monitor only if its priority is higher than the highest priority ceiling of all monitors that have been entered by the other jobs.

Consequently, the 3 main points of this paper (as per the rubric) are that:

- A set of n periodic tasks using the priority ceiling protocol can be scheduled if the priorities are based on RM theory and that the following conditions are met:

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

- The priority inversion problem can be solved by the *priority inheritance protocol* but this can lead to Direct blocking and Push Through Blocking. This can be further avoided by the *priority ceiling protocol* which has its own blocking problem, referred to as *Ceiling blocking*, but as demonstrated in the latter half of the paper, the blocking time is still far better compared to the blocking times of Direct and Push Through Blocking.
- Finally, all of these blocking issues originate from the idea that resources are being shared and synchronization primitives such as semaphores, monitors or the Ada rendezvous are being used in order to synchronize multiple processes/jobs.

“Dr. Siewart’s summary for Resource Management”:

This paper continues to discuss that when using SoC principles, the goal for the processor should not be to “be sized not only based upon clock rate or benchmark MIPS (millions of instructions per second), but rather on efficiency in execution of services.” It then continues to breakdown the SoC into its three major resources - data movement, computation, and data storage, i.e. IO, CPU and memory. Dr. Siewart also talks about priority inversion and the “five 9s” in which he says that a “five nines” system i.e. one with 99.999% uptime isn’t allowed to be out of service for more than 5 minutes a year and so rebooting the system is not a viable option. He then discusses the three conditions for priority inversion:

- 3 or more services in the system sharing a CPU
- 2 services with priority H and L accessing a shared resource
- $\text{Prio}(H) > \text{prio}(M1) > \text{prio}(M2) > \text{prio}(L)$.

He states that if a software service designer can avoid any of the above conditions, then an unbounded priority inversion will not arise.

He continues to give an overview of deadlock and an example while also providing three solutions to recover from the deadlock:

- Avoidance: Multiresource acquisitions are strictly serialized to preclude circular wait
- Detection by monitor with random back-off: Resources and monitors are dropped and acquired at a random delay which is increasing in the amount of time.
- Detection by hardware Watchdog: Failure to provide service is detected by some hardware timer and the system is rebooted to start recovery.

Why the Linux position makes sense or not

So we’re given two opposing views to read and conclude which one makes more sense and to validate whether the Linux Position on Priority Inheritance makes sense or not. Linus, the father of Linux, was fairly clear in how he felt on this subject when he said “Friends don’t let friends use priority inheritance” and “Just don’t do it. If you really need it, your system is broken anyways.” Ingo Molnar, a member of the Kernel Development community, however, believed that the idea of locking could not be ignored and the only alternative would be very convoluted and tedious code. As such, he requested a priority-inheritance futex implementation for the kernel.

Now, I personally think that Linus’s position towards Priority Inheritance protocols is a bit too extreme because it doesn’t allow for cases which are only solvable using shared resources and for certain tasks to be prioritized above others based on their criticality. Specifically, let’s discuss the Mars example that was referenced in class - there are multiple tasks that the spacecraft will be conducting but in the case of imaging and attitude control, certain tasks will be more critical

to be completed on time compared to other. I.e. it's more important the spacecraft is in the right attitude. Consequently, in order to ensure this and not use Priority Inheritance protocols, it would require very convoluted software development and be near impossible to ensure that execution is reproducible. Consequently, although I am able to understand the reasoning behind Linus's position, it doesn't make sense for many complex applications.

Reasoning on whether Futex fixes unbounded priority inversion

A priority inheritance futex - as implemented by Ingo Molnar - is an attempt to solve the unbounded priority inversion issue because unlike traditional mutexes, which are implemented in just the kernel space, Priority Inheritance Futexes can be taken without involving the kernel at all treating it like a normal futex and therefore minimizing the overhead time. Instead, the kernel only needs to know about a PI-futex while it is being contented meaning that the number of futexes can get quite large before we have to start worrying about overhead on the kernel side. Additionally, priority inheritance is chained so that if the "holding process is blocked on a 2nd futex, the boosted priority will propagate to the holder to that second futex and as soon as a futex is released, any associated priority boost will be removed as well." What all this means is that in Priority Inheritance Futexes, if the priority of a task blocking the current task is lower, there is a mechanism of "boost" which boosts the original priority inherited by the additional futex and would therefore resume original execution once the current task has completed executing. Ultimately, this resolves the issue of a task having to wait indefinitely when a lower priority task acquires a resource and therefore solves the unbounded priority inversion issue.

Problem 2

[25 points] Review the terminology and describe clearly what it means to write "thread safe" functions that are "re-entrant". There are generally three main ways to do this: 1) pure functions that use only stack and have no global memory, 2) functions which use thread indexed global data, and 3) functions which use shared memory global data, but synchronize access to it using a MUTEX semaphore critical section wrapper. Describe each method and how you would code it and how it would impact real-time threads/tasks. Now, using a MUTEX, provide an example using RT-Linux Pthreads that does a thread safe update of a complex state with a timestamp (pthread_mutex_lock). Your code should include two threads and one should update a timespec structure contained in a structure that includes a double precision attitude state of {X,Y,Z acceleration and Roll, Pitch, Yaw rates at Sample_Time} (just make up values for the navigational state and see http://linux.die.net/man/3/clock_gettime 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:

A “thread-safe” function is one which guarantees safety of variables even if it is being used by multiple threads at the same time, whereas a “reentrant” function is one which, even if interrupted during execution, can be resumed by the interrupting task without interfering with the previous course of execution.

In order to write a thread-safe reentrant function, we need to make sure that :

- a. To ensure thread-safety, either thread-local variables are used, or no global variables are used.
- b. To ensure reentrancy, if global variables are used, they are restored within the context of the thread before thread exits.

1. Pure functions which use only stack and have no global memory

The least complex method of execution, and the ideal method to make sure that functions are both reentrant as well as thread-safe. However, it may often be impractical, as global data may sometimes be desired. This method can be coded by using only local variables and ignoring global variables for the scope of the function which we wish to make reentrant thread-safe.

If we consider a simple example code of swapping two numbers, we wouldn't use any global variables to temporarily hold the value of one variable. Instead a local variable would be initialized and set to one swap value, while the numbers are swapped.

As for how this method would impact real-time tasks, this method is of course, the ideal solution as far as thread-safety is concerned. Two disadvantages of using this method would be that if too many local variables are created, the thread would run out of stack memory, and stack memory is generally not available in large amounts. The other disadvantage is that of resource sharing between threads. Without the use of global data, this would have to be done by passing thread parameters, which introduces its own complexities. Of course, they can be worked around to have the advantage that pure functions provide, that of unchangeability by unaccounted-for factors.

2. Functions which use thread indexed global data

A convenient method of using global variables for particular threads, by indexing those variables for their respective threads. This method can be executed by making a variable thread-local at the time of its global declaration. It is allocated an index, and if any thread must use the variable, it must do so by accessing the index, and storing a pointer to a copy of the variable dynamically.

If we consider the same example code of swapping two numbers, this thread-local variable could be used to temporarily hold the value of one of the numbers to be

swapped. This particular method ensures thread-safety as well as reentrancy from the function being called within the same thread.

As to the method's impact on real-time tasks, it certainly solves one of the problems posed by the earlier method, that of stack memory for each thread. With the variables being declared globally, we no longer have to worry about that, and the variables are owned by the task context and stored in the Task Control Block. The major problem with this method is that it allocated memory dynamically, in the heap. Allocation of memory is a relatively time-consuming task, and can slow down execution of the program. Care must also be taken to free the same memory at the time of exit.

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

Although the most complex method, it is also the most elegant and widely used method. It uses global variables, but uses either a mutex or a semaphore to restrict any other thread from accessing those variables at the same time. These sections of code are called critical sections, and are best protected by the use of such mechanisms. This method can be executed by declaring variables globally, and using them within threads, but by wrapping their use in a "mutex_lock" or by synchronizing threads such that other threads wait for the semaphore value to be incremented, before being allowed access to the variable.

In the situation of real-time tasks, using mutexes and semaphores can sometimes lead to unexpected deadlocks, if improperly synchronized, but in general these mechanisms are quite useful. They solve both issues posed by the 1st method, they are declared globally, so do not use up the stack memory, and all threads can use the global data if required.

The only possible disadvantage to using them is that they can be considered as 'blocking' mechanisms. To clarify, if one thread locks a particular piece of code, and another thread tries to access that same piece of code, it would be blocked from doing, and would have to wait till the first thread gives up control. It is a relatively small disadvantage, but one to be considered nevertheless.

Synchronization using Mutex and RT-Linux pthreads

Synchronization of two threads using MUTEXes is done in our example code. The first code updates a structure containing six precision values, and a timestamp value. The thread protects this update by using a mutex, hence making sure no other thread can access the data during update, which might have led to possible corruption. The second thread simply reads the data provided by the structure, and prints the same, along with the current timestamp, proving the data has not been corrupted.

A screenshot of the output is provided below:

```
thread_safe thread_safe.c
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/thread_safety$ ./thread_safe

Updating Thread
Updated Timestamp 1552021631 secs, 898363768 nsecs
Updated acceleration is 466407008.217188 in x, 1754918656.817198 in y, 1529358720.712163 in z
Updated attitude is roll of 861744704.401281, pitch of 237986464.110821, yaw of 1726082816.803770

Reading Thread Time instant is 1552021632 secs, 897199466 nsecs
Timestamp obtained from above is 1552021631 secs, 898363768 nsecs
Acceleration obtained is 466407008.217188 in x, 1754918656.817198 in y, 1529358720.712163 in z
Attitude obtained is roll of 861744704.401281, pitch of 237986464.110821, yaw of 1726082816.803770

-----

Updating Thread
Updated Timestamp 1552021633 secs, 918517640 nsecs
Updated acceleration is 1556157312.724642 in x, 451651680.210317 in y, 1821956864.848415 in z
Updated attitude is roll of 1807021568.841460, pitch of 769780416.358457, yaw of 1579111936.735331

Reading Thread Time instant is 1552021635 secs, 898182644 nsecs
Timestamp obtained from above is 1552021633 secs, 918517640 nsecs
Acceleration obtained is 1556157312.724642 in x, 451651680.210317 in y, 1821956864.848415 in z
Attitude obtained is roll of 1807021568.841460, pitch of 769780416.358457, yaw of 1579111936.735331

-----

Updating Thread
Updated Timestamp 1552021635 secs, 922590362 nsecs
Updated acceleration is 54298344.025285 in x, 19668744.009159 in y, 1524438784.709872 in z
Updated attitude is roll of 1330274688.619457, pitch of 447741984.208496, yaw of 1936376960.901696

Updating Thread
Updated Timestamp 1552021637 secs, 926294679 nsecs
Updated acceleration is 1048110272.488064 in x, 1880380032.875620 in y, 1016710080.473443 in z
Updated attitude is roll of 1469482880.684281, pitch of 222978112.103832, yaw of 234776640.109326

Reading Thread Time instant is 1552021638 secs, 901678192 nsecs
Timestamp obtained from above is 1552021637 secs, 926294679 nsecs
Acceleration obtained is 1048110272.488064 in x, 1880380032.875620 in y, 1016710080.473443 in z
Attitude obtained is roll of 1469482880.684281, pitch of 222978112.103832, yaw of 234776640.109326

-----
```


Problem 3

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

Solution:

1. Issue of deadlock

The 'deadlock.c' code has two threads attempting to access two resources in reverse order. Thread 1 locks Resource A first, then attempts to lock Resource B. Similarly, thread 2 locks Resource B first, then Resource A. Obviously, both threads will lock Resource A and Resource B respectively at roughly the same time, and will arrive at the instruction 'pthread_mutex_lock(rsrcA/rsrcB)', and will find the resource they are trying to lock already locked by the other thread. Since 'pthread_mutex_lock' is a blocking function, both threads wait at this point for their respective resource to be released. This leads to a deadlock in a program. For the purpose of resolving this deadlock, I have used a random sleep time for thread 1, and the function 'pthread_mutex_trylock' for thread 2's locking of resource B. How this solves our problem is by making the threads sleep for a random amount, achieving the 'random' section of random back-off. The trylock function helps in the backoff section. The function has basically the same purpose as 'pthread_mutex_lock', but instead of blocking the call, it returns an error value if the resource has already been locked by another thread. The code provided has three conditions, race, safe and unsafe. The safe condition blocks a particular thread for some time, allowing the other thread to run to completion first. The race condition launches both threads, creating a race of sorts between the two threads to acquire both mutex locks before the other. The unsafe condition puts both threads to sleep, ensuring a deadlock condition.

Exercise 3

A screenshot of the safe and race conditions is provided below:

```
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$ ./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 2 got B, trying for A
THREAD 2 got B and A on attempt 1
THREAD 2 done
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B on attempt 2
THREAD 1 done
Thread 1: -1226324896 done
Thread 2: -1234713504 done
All done
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$ ./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 on attempt 1
THREAD 1 done
Thread 1: -1226611616 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 2 got B, trying for A
THREAD 2 got B and A on attempt 2
THREAD 2 done
Thread 2: -1226611616 done
All done
```

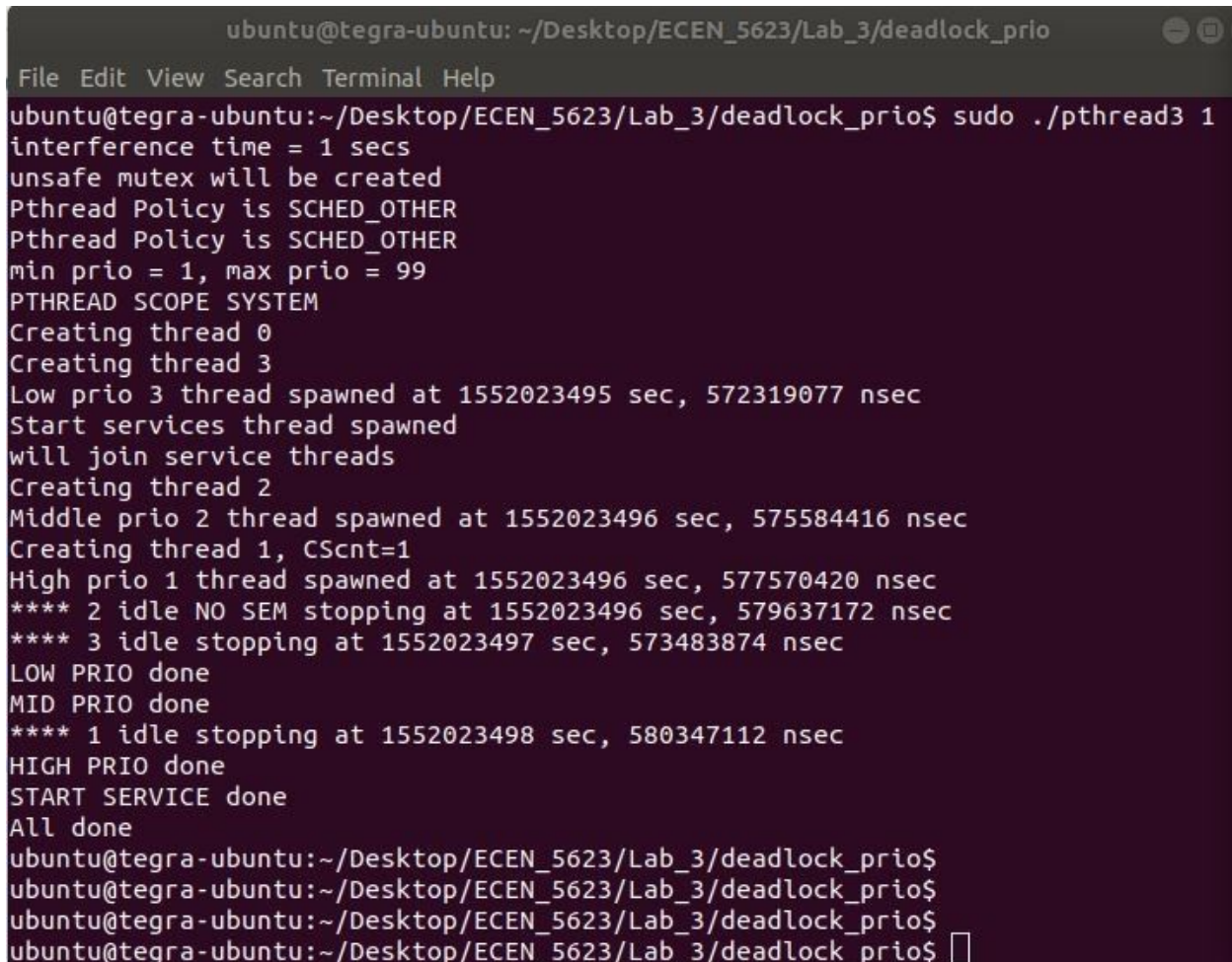
Below is a screenshot of the unsafe condition, and our implementation to resolve the same, using a random sleep time in the range of microseconds, and the trylock function:

```
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$ ./deadlock_unsafe
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
^C
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$ ./deadlock
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
rsrcACnt=0, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 grabbing resources
THREAD 2 got B, trying for A
Task 2 deadlocked, resetting mutex locks
THREAD 2 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B on attempt 1
THREAD 1 done
Thread 1: -1226972064 done
THREAD 2 got B, trying for A
THREAD 2 got B and A on attempt 3
THREAD 2 done
Thread 2: -1235360672 done
All done
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$
```

2. Issue of Priority inversion

Priority inversion means that in a scenario of three threads, a thread of lower priority manages to complete its task before the higher priority thread. Let us examine the step-wise occurrence of this phenomenon.

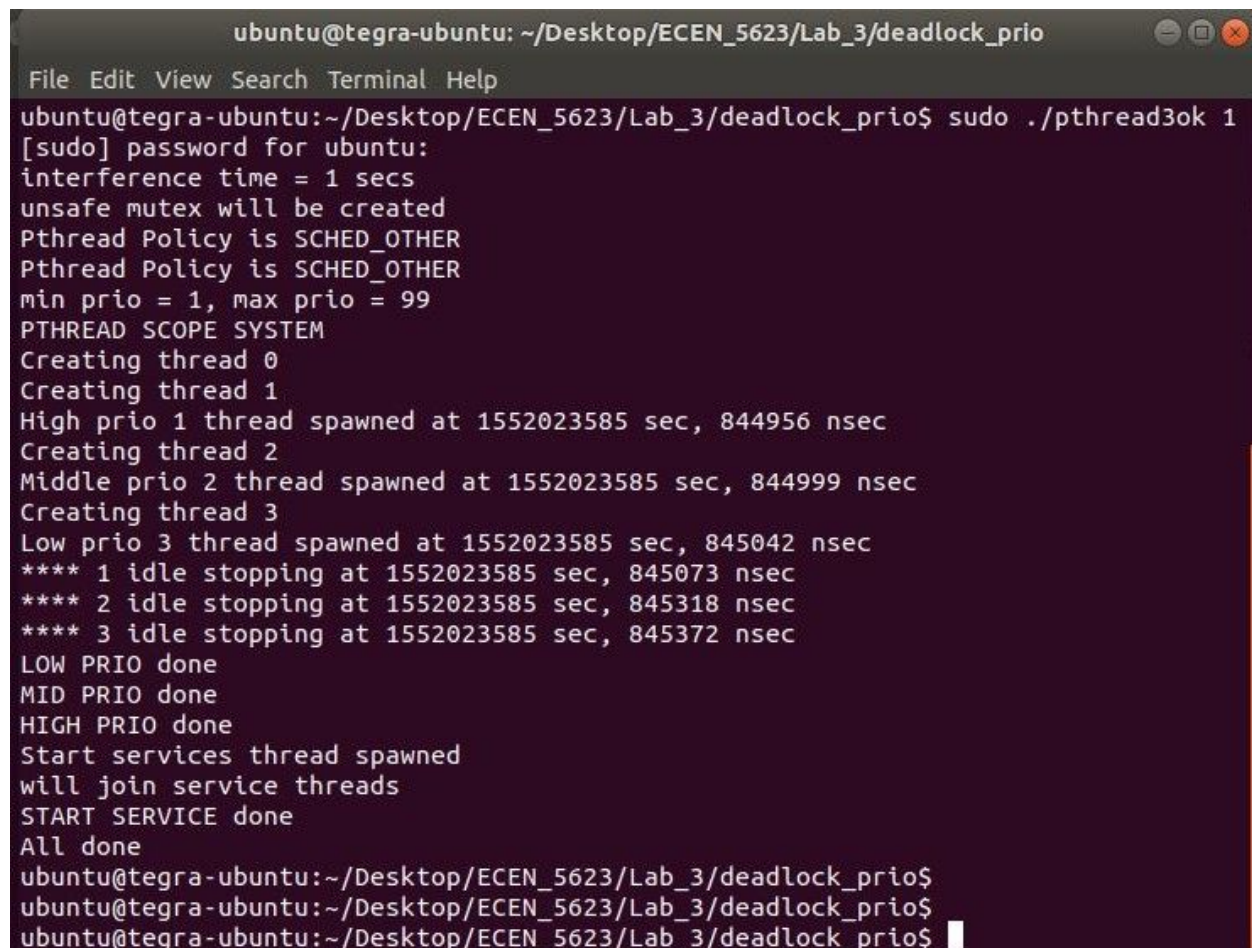
If we consider a task with priority L running on a thread, which has locked a mutex, and another task with priority $H > L$ on another thread, where H also attempts to lock the same mutex resource, L will not allow H to preempt it before completing its critical section. Now during this execution, if another task of priority M, where $H > M > L$, is spawned on another thread, M will preempt L, and complete its execution before both L as well as H. This phenomenon of M running its course before H in spite of not being in control of a critical section is called priority inversion. We see exactly this same phenomenon in the example code provided. An output screenshot is provided below:



```
ubuntu@tegra-ubuntu: ~/Desktop/ECEN_5623/Lab_3/deadlock_prio
File Edit View Search Terminal Help
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$ sudo ./pthread3 1
interference time = 1 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Creating thread 3
Low prio 3 thread spawned at 1552023495 sec, 572319077 nsec
Start services thread spawned
will join service threads
Creating thread 2
Middle prio 2 thread spawned at 1552023496 sec, 575584416 nsec
Creating thread 1, CSnt=1
High prio 1 thread spawned at 1552023496 sec, 577570420 nsec
**** 2 idle NO SEM stopping at 1552023496 sec, 579637172 nsec
**** 3 idle stopping at 1552023497 sec, 573483874 nsec
LOW PRIO done
MID PRIO done
**** 1 idle stopping at 1552023498 sec, 580347112 nsec
HIGH PRIO done
START SERVICE done
All done
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$
```

Although the three threads are spawned together, M completes executing first.

To solve this, we can use either priority ceiling or priority inheritance. Priority inheritance occurs when if L has locked a critical section, and the critical section is shared by another task of higher priority H, task L is temporarily assigned the higher priority of H. A screenshot of the code where priority inheritance was employed to solve the above problem is attached below:



```
ubuntu@tegra-ubuntu: ~/Desktop/ECEN_5623/Lab_3/deadlock_prio
File Edit View Search Terminal Help
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$ sudo ./pthread3ok 1
[sudo] password for ubuntu:
interference time = 1 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Creating thread 1
High prio 1 thread spawned at 1552023585 sec, 844956 nsec
Creating thread 2
Middle prio 2 thread spawned at 1552023585 sec, 844999 nsec
Creating thread 3
Low prio 3 thread spawned at 1552023585 sec, 845042 nsec
**** 1 idle stopping at 1552023585 sec, 845073 nsec
**** 2 idle stopping at 1552023585 sec, 845318 nsec
**** 3 idle stopping at 1552023585 sec, 845372 nsec
LOW PRIO done
MID PRIO done
HIGH PRIO done
Start services thread spawned
will join service threads
START SERVICE done
All done
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/deadlock_prio$
```

3. RT_PREEMPT patch, and whether simply switching to RTOS over Linux makes sense

Firstly, there is no real way of fixing priority inversion in Linux without using the RT_PREEMPT patch. Linux just does not support true real-time services.

The advantage of using a RT_PREEMPT patch is that converts Linux into a preemptible kernel. It does so by quite a few mechanisms, using spin locks, critical section locks like mutexes and semaphores and converting interrupt handlers into preemptible kernel threads. The effect of this is that the user can basically change the priority of any thread he wishes to. While this is not a

Exercise 3

sure shot solution, and does not guarantee that priority inversion will never occur, it certainly ensures that unbounded priority inversion will not occur.

As for whether switching to an RTOS over Linux makes sense, it vastly depends on the application. Of course, Linux will never be a completely real-time OS, regardless of any patches one may use. However, the overwhelming advantage Linux carries over RTOS is the large volume of pre existing code for virtually any application in Linux. It is a well-known fact that using tested code is a better practice than writing new code. Linux is the brainchild of that fact. Keeping the above in mind, it is worth noting that if an application requires a hard real-time OS, and the industry behind the application is willing to pour resources into the same, it might be worth implementing the required set of services on an RTOS instead of Linux. There is no single answer to this question. RTOS has its own set of strengths, and so does Linux.

Problem 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?

Solution:

So we ported the VxWorks code over to the Tegra TK1 using the RT-linux FIFO pthreads (similar to what we used in previous examples) instead of the traditional VxWorks task. It must be noted that we used the provided `heap_mq.c` and `posix_mq.c` for inspiration so the receive and transmit methods translate over fairly well and we've also kept the message queue name the same, except that we instead appended a '/' at the beginning in order to avoid a message queue initialization error. With this change, we then implemented two threads - i.e. rx and tx - which modeled the VxWorks tasks and gave the receiver task a higher priority than the sender task (i.e. 99 vs 98). The message queues were then assigned the appropriate information such as the maximum size (128 from VxWorks), `max_messages` and any necessary flags (n/a). The receiver thread was then created first because it has a higher priority and this thread creates a message queue and waits to receive on that message queue. This thread will continue to wait for as long the message queue received any data and so until it reaches that state, it is effectively blocked.

The tx thread is then created and this thread also opens the same message queue and then sends a predetermined message. The message was copied over from the VxWorks example.

Note that one of the major hurdles that we had to address was that POSIX messages are not enabled by default in the Jetson TK1 kernel. As such, we had to update the kernel to enable the POSIX messages and then rebuild it and boot into it in order to use the `mq_send` and `mq_receive` functions. Prior to completing this step, we kept getting an error indicating that the functions `mq_send` and `mq_receive` were not implemented.

Finally, implementing this, we got the following results:

```
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/problem_4$ make clean
rm -f *.o *.d
rm -f posix_new
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/problem_4$ make
gcc -O0 -g -c posix_new.c
gcc -O0 -g -o posix_new posix_new.o -lm -lrt -lpthread -std=gnu99
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/problem_4$ sudo ./posix_new
[sudo] password for ubuntu:
RX Method:
RX Thread receiving from Message Queue
TX createdTX Method:
TX Thread sending to Message Queue
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
Sent TX Message
TX createdubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/problem_4$
```

Therefore, as it can be seen, we were able to successfully create the two threads (tx and rx) and write/read a message from the message queue.

Now, implementing the heap code on the Jetson TK1, we largely followed Professor Siewart's VxWorks example once again but adapted it to the POSIX scheduling policies. Specifically, similar to the posix_new example, we setup two threads (an rx thread and a tx thread), created a message queue and set all that up in a similar fashion. The once change we implemented compared to the posix_new example was that the contents that we were sending were stored in the heap. In this case, the VxWorks example had stored an imagebuffer in the heap and a pointer was declaring to that buffer and therefore, that's similar to what we implemented. The threads were then given the appropriate priorities and the rx thread waited for the tx thread to send the contents of the heap before continuing at the point of execution. Notice below that the output is also repeating and this was simply because we had a while loop with the same amount of delay as was implemented in the VxWorks - 3 seconds.


```

ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/problem_4$ ls -lrt
total 56
-rw-r--r-- 1 ubuntu ubuntu 3430 Mar  8 07:17 posix_new.c
-rw-r--r-- 1 ubuntu ubuntu  476 Mar  8 07:19 Makefile
-rw-r--r-- 1 ubuntu ubuntu 4503 Mar  8 07:21 heap_new.c
-rw-rw-r-- 1 ubuntu ubuntu 7052 Mar  8 07:21 posix_new.o
-rwxrwxr-x 1 ubuntu ubuntu 11733 Mar  8 07:21 posix_new
-rw-rw-r-- 1 ubuntu ubuntu 8120 Mar  8 07:21 heap_new.o
-rwxrwxr-x 1 ubuntu ubuntu 12112 Mar  8 07:21 heap_new
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/problem_4$ sudo ./heap_new
[sudo] password for ubuntu:
buffer =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~ RX Method:
Reading 4 bytes
RX createdTX Method:

Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 4 bytes
receive: ptr msg 0xB5D00468 received with priority = 30, length = 8, id = 99
contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
heap space memory freed
Reading 4 bytes
send: message ptr 0xB5D00468 successfully sent
TX created
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 4 bytes
receive: ptr msg 0xB5D00468 received with priority = 30, length = 8, id = 99
contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
heap space memory freed
Reading 4 bytes
send: message ptr 0xB5D00468 successfully sent

Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 4 bytes
receive: ptr msg 0xB5D00468 received with priority = 30, length = 8, id = 99
contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
heap space memory freed
Reading 4 bytes
send: message ptr 0xB5D00468 successfully sent
^Cubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/problem_4$ █

```

Finally, I agree that message queues circumvent the issue of unbounded inversion because they are implemented as to synchronize the passage of messages between various threads and tasks of execution. Specifically note that if we have global memory sharing, as was the case in this problem, we can use message queues in order for threads to be pause until a message is passed to the queue before continuing on with execution. In the case of the tx thread, if the message queue is already completely full, it will be blocked until a rx thread is able to read some of the

messages at which point there will more space available for the tx thread to transmit the messages.

They additionally address the issue of priority inversion because often times, message queues have priorities associated with them meaning that higher priority messages get dequeued prior to the lower priority messages. This, in turn, means that lower priority messages cannot preempt a higher priority message and therefore priority inversion would not be possible.

Problem 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.

Solution:

Use of watchdog timers to solve deadlocks:

The essence of the watchdog timer is to watch the system for nominal behavior and minimize the impact of any anomalous behavior on the system. It does this as being the “last line of defence” for the system by simply resetting the system in order to return it to a known operational state. It can be thought of as the equivalent of turning the system off and back on again. This process, however, does not add any insight into what the issues in the system may be but instead focuses on not letting the issues propagate to causing any unwanted behavior. As described by the online resource provided, the linux watchdog is configured to have been broken into two distinct parts: the Watchdog Daemon and the “Watchdog Device”.

It was described in the online resource that daemon itself would be fairly useless without the device and so it’s important to talk about that first. The “Watchdog Device” is simply a hardware timer which will, upon time-out, reset the computer in the same manner as a nominal hardware reset switch. Because this timeout is a fixed value, it should be reset when operations are executing nominally as to not trip upon non-fatal operations. In the linux kernel, the watchdog device is linked as a driver module in the `/dev/watchdog` module and the `watchdog-timeout` value can also be updated (but highly discouraged).

There is also a watchdog daemon in Linux which runs in the background continually refreshing the timer. If, for instance, a process limits the daemon to be able to refresh the timer, then after the specified timeout value, the system will reboot itself. It must be noted that this daemon actually tries to lock itself into non-paged memory and sets its priority to realtime meaning that it attempts to function as far as possible even if the machine load is very high.

As a consequence, it can be useful in the cases of a system deadlock because the deadlock would block the other process from being able to reset the watchdog timer resulting in the timer to be tripped and for the system to reset itself. This would prevent the case for the system to stay in an indefinite deadlock and would instead just be blocked for the duration of the watchdog timer.

Modified code from Problem #2

In the original code from problem 2, I added another thread, to try and access the mutex resource. Obviously, the 'updating' thread has the mutex locked for a majority of the time, which is going to lead to a deadlock when the 'reading' thread tries to access the mutex. To solve this deadlock and implement our own version of a watchdog timer, I used the function 'pthread_mutex_timedlock'. The function blocks for an amount of time specified by the 'struct timespec' argument, after which it returns an error value, if it has still not been able to acquire the mutex lock. A print statement saying 'No new data available' is implemented whenever the timed_lock mutex times out. Attached below is the output screenshot:

```
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Lab_3/timedlock_mutex$ ./timed_lock

No new data available at time 1552084728 secs, 852896019 nsecs

-----

Updating Thread
Updated Timestamp 1552084729 secs, 851211733 nsecs
Updated acceleration is 1447410432.674003 in x, 939508224.437493 in y, 1416848000.659771 in z
Updated attitude is roll of 817230080.380552, pitch of 259111600.120658, yaw of 1469832832.684444

-----

Reading Thread Time instant is 1552084730 secs, 850927024 nsecs
Timestamp obtained from above is 1552084729 secs, 851211733 nsecs
Acceleration obtained is 1447410432.674003 in x, 939508224.437493 in y, 1416848000.659771 in z
Attitude obtained is roll of 817230080.380552, pitch of 259111600.120658, yaw of 1469832832.684444

-----

No new data available at time 1552084738 secs, 853630399 nsecs

-----

Updating Thread
Updated Timestamp 1552084741 secs, 853270467 nsecs
Updated acceleration is 659642240.307170 in x, 1522741632.709082 in y, 1063808256.495374 in z
Updated attitude is roll of 589783616.274639, pitch of 586487616.273105, yaw of 1519799296.707712

-----

Reading Thread Time instant is 1552084743 secs, 851074443 nsecs
Timestamp obtained from above is 1552084741 secs, 853270467 nsecs
Acceleration obtained is 659642240.307170 in x, 1522741632.709082 in y, 1063808256.495374 in z
Attitude obtained is roll of 589783616.274639, pitch of 586487616.273105, yaw of 1519799296.707712

-----

No new data available at time 1552084748 secs, 853751618 nsecs

-----
```

References

- [1] "Priority Inheritance Protocols: An Approach to Real-Time Synchronization"-
http://mercury.pr.erau.edu/~siewerts/cec450/documents/Papers/prio_inheritance_protocols.pdf
- [2] "Writing Reentrant and Thread-safe functions"-
https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/com.ibm.aix.genprogc/writing_reentrant_thread_safe_code.htm
- [3] "Priority inheritance in the kernel" - <https://lwn.net/Articles/178253/>
- [4] "Build your own Jetson TK1 kernel" -
<https://devtalk.nvidia.com/default/topic/762653/-howto-build-own-kernel-for-jetson-tk1/>
- [5] Message Queues - <https://www.geeksforgeeks.org/ipc-using-message-queues/>
- [6] Linux Watchdog Daemon -
<http://www.sat.dundee.ac.uk/psc/watchdog/watchdog-configure.html>
- [7] Real time kernel patch set
https://wiki.archlinux.org/index.php/Realtime_kernel_patchset