July 7, 2017

ECEN 5623 – Real Time Embedded Systems

Summer 2017

EXERCISE 3

Threading/Tasking and Real-Time Synchronization

Jeet Baru

Email: jeet.baru@colorado.edu

ID Number: 107189037

Question 1

In this question we analyze the paper on priority inheritance protocol and and summaries written for the paper. We understood the paper and make list the 3 key points stated as:

1. Using semaphores for synchronization shared memory access for tasks in priority driven preemptive real time systems can have blocking issues where tasks of lower priority may block a high priority task for indefinite period of time and to solve this problem, priority inheritance techniques should be used.
2. The basic priority inheritance resolves the unbounded blocking by letting 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 the higher priority task can be blocked a maximum of m times if there are m semaphores. Still, the basic priority inheritance cannot prevent deadlocks and also the unbounded blocking period can be significant due to chain of blocking by the other tasks.
3. The priority ceiling protocol, in addition to bounding the blocking period, also prevents deadlocks and chained blocking. With the basic priority inheritance for the tasks, it also assigns a priority ceiling to each semaphore. Thus an inherited higher priority task blocks a high priority task while sharing multiple semaphores such that the inherited task gets all the required semaphores, effectively avoiding deadlocks and chained blocking. This significantly reduces the blocking period to only one critical section.

Why the Linux Position Makes Sense

In context of using priority inheritance to avoid priority inheritance, I would be in favor of using priority inheritance instead of not using it as mandated by Linus Torvalds. It is true that for completely fair scheduling used in desktop operating systems today, there cannot be priority inversion as tasks aren't pre-empted and all tasks run for a fair share of time. But for SCHED_FIFO or RTOS, higher priority tasks will preempt lower priority task and thus this may lead to tasks wanting to access shared memory simultaneously, which causes priority inversion. As the number of processes using the shared memory increases, the probability of blocking increases and this unbounded blocking can easily cause system failure. To prevent this blocking, priority inheritance can be used so that lower priority tasks can use inherited priority to run as a high priority task. Even any occurring deadlocks can be prevented using the priority ceiling protocol, but using these methods would often be very complicated and slow down the locking code. Thus they would always be rejected from being merged into the Linux kernel code. Linux is in the market of user end operating systems and hence its position is vindicated, as far as hard real time applications are concerned Linux is not essentially designed for those.

Futexes as a solution for Unbounded Inversion

Futexes are fast locking mechanisms used in userspace. Ingo Molnar developed priority inheritance support for futexes. As futexes are operated in user space, they are can be easily modified using an atomic operation like CMPXCHG in x86. All this happens in the userspace and the kernel maintains no information about the lock state. In userspace applications cannot use spinlocks to disable interrupts or make a task non-preemptable in a critical section. Thus to ensure deterministic scheduling for userspace applications, priority inheritance has to be used. In case of no contention for accessing shared memory, PI-futex does not make any kernel calls. In case there is a contention the FUTEX_LOCK_PI operation is requested from the kernel. The kernel will do the rest of the job, i.e., using priority inheritance protocols. Thus the kernel effectively knows about PI-futex only if there is any contention.

The PI-futex works faster compared to the kernel space mutex implementation as kernel level calls are reduced in cases of no contention. The Native POSIX Thread Library (NPTL) uses PI-futexes for the pthread mutex implementation in userspace. They provide safe and accurate protection from unbounded priority inversion in the userspace, but it may still have unbounded inversion by blocking in-kernel mutexes, spinlocks or semaphores.

Question 2

Thread Safe Functions

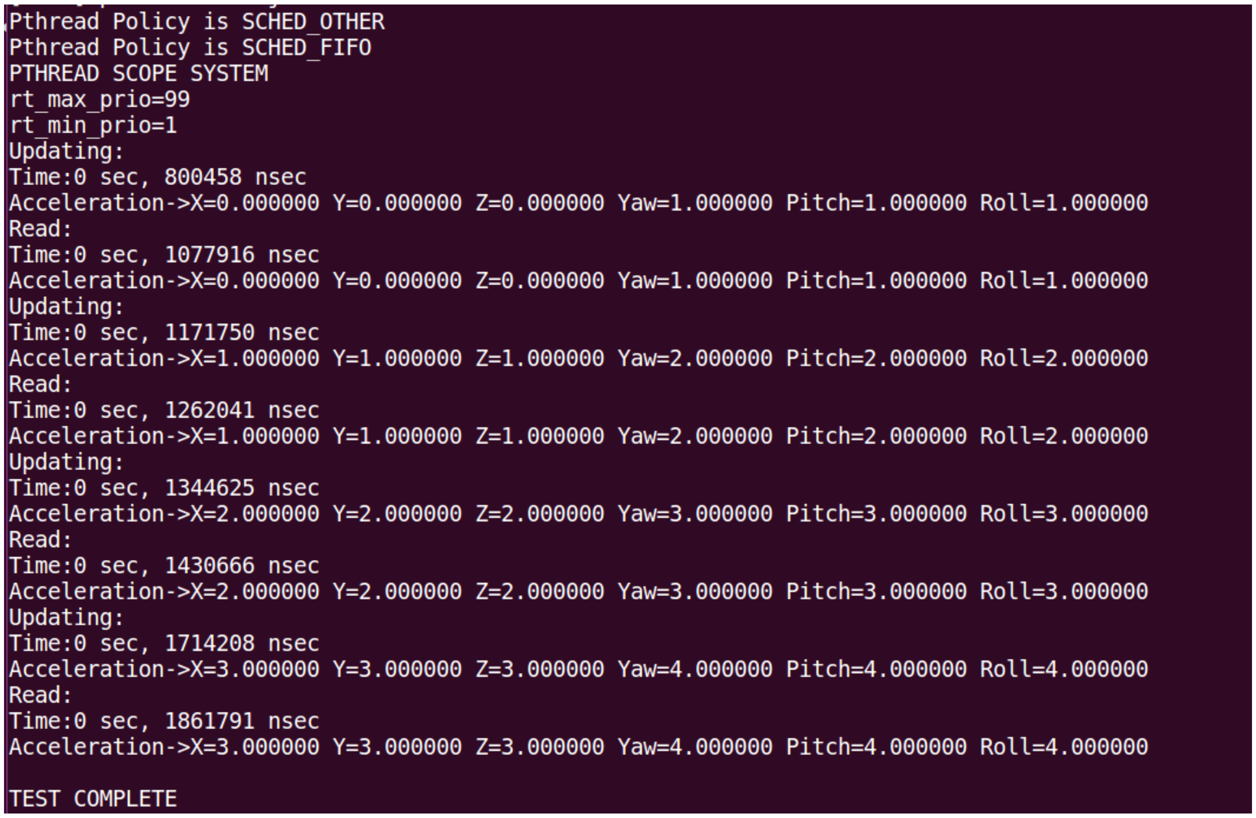
If one function may be called by more than one thread and those threads are concurrently active, the shared function must be written to provide reentrant function support so that it is thread-safe. If the function updates some kind of global data then it can be a problem as both the threads can corrupt that data. Hence in such a condition it is very important to make the thread safe such that so it is not interfered by other thread when it is updating the common global data. There are basically 3 ways of doing this-

1. To use pure functions that use only stack and have no global memory. This is the easiest way to ensure that there are no chances of data corruption because there is no data which can be shared between 2 threads which can be corrupted. Each function will only have its own local data which it can change on its own. However, this is doesn't solve the problem because many times we require sharing of data between real time threads. Coding this is very similar to writing simple function with its own local data.

2. Functions which use thread indexed global data can be a very useful way to ensure that thread remains safe as well as they can have global variables to share the data. The concept of thread indexed global data is that it defines a mechanism whereby code can retrieve thread-specific data stored in a global database accessed by a all-threads-known shared key. Therefore when a thread requests this data using the key, they will all get back the address. I can then use this address to access the data. This is a way for threads to communicate for real time systems but still mutexes and semaphores are better suited.

3. Using functions which use shared memory global data but synchronize access to it using a MUTEX semaphore critical section wrapper. This is best way to code any real time threads which require to share data between them without corrupting it. We can create a mutex globally and it can be locked by threads in the critical sections where they update or read any global data, which ensures that the are not interrupted by any other thread during critical update and read. And when in a function we ensure that update and read are safe of one another, then we can be sure that the data will not be corrupted. We can use the mutex_init to first initialize a mutex in the main thread. Then the corresponding threads can use mutex_lock() and mutex_unlock() functions around the critical section to make section safe.

Now we write a code implementing a thread safe application. Here we use RT-Linux pthread api to design an application that modifies contents of a structure with timestamps and read the changes without corrupting the global data.



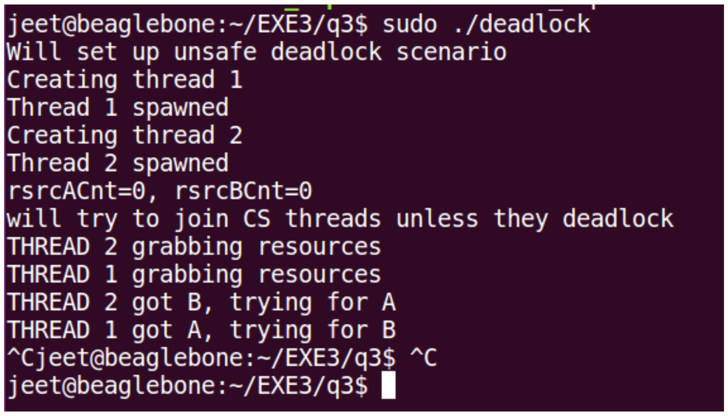
```
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1
Updating:
Time:0 sec, 800458 nsec
Acceleration->X=0.000000 Y=0.000000 Z=0.000000 Yaw=1.000000 Pitch=1.000000 Roll=1.000000
Read:
Time:0 sec, 1077916 nsec
Acceleration->X=0.000000 Y=0.000000 Z=0.000000 Yaw=1.000000 Pitch=1.000000 Roll=1.000000
Updating:
Time:0 sec, 1171750 nsec
Acceleration->X=1.000000 Y=1.000000 Z=1.000000 Yaw=2.000000 Pitch=2.000000 Roll=2.000000
Read:
Time:0 sec, 1262041 nsec
Acceleration->X=1.000000 Y=1.000000 Z=1.000000 Yaw=2.000000 Pitch=2.000000 Roll=2.000000
Updating:
Time:0 sec, 1344625 nsec
Acceleration->X=2.000000 Y=2.000000 Z=2.000000 Yaw=3.000000 Pitch=3.000000 Roll=3.000000
Read:
Time:0 sec, 1430666 nsec
Acceleration->X=2.000000 Y=2.000000 Z=2.000000 Yaw=3.000000 Pitch=3.000000 Roll=3.000000
Updating:
Time:0 sec, 1714208 nsec
Acceleration->X=3.000000 Y=3.000000 Z=3.000000 Yaw=4.000000 Pitch=4.000000 Roll=4.000000
Read:
Time:0 sec, 1861791 nsec
Acceleration->X=3.000000 Y=3.000000 Z=3.000000 Yaw=4.000000 Pitch=4.000000 Roll=4.000000

TEST COMPLETE
```

As seen above 2 threads were created one updates values of all the parameters and the timestamp in the global structure and 2nd one reads the same structure and prints out the entire data. Both the threads use a global mutex while updating as well as reading so that the data is never corrupted and the threads remain safe.

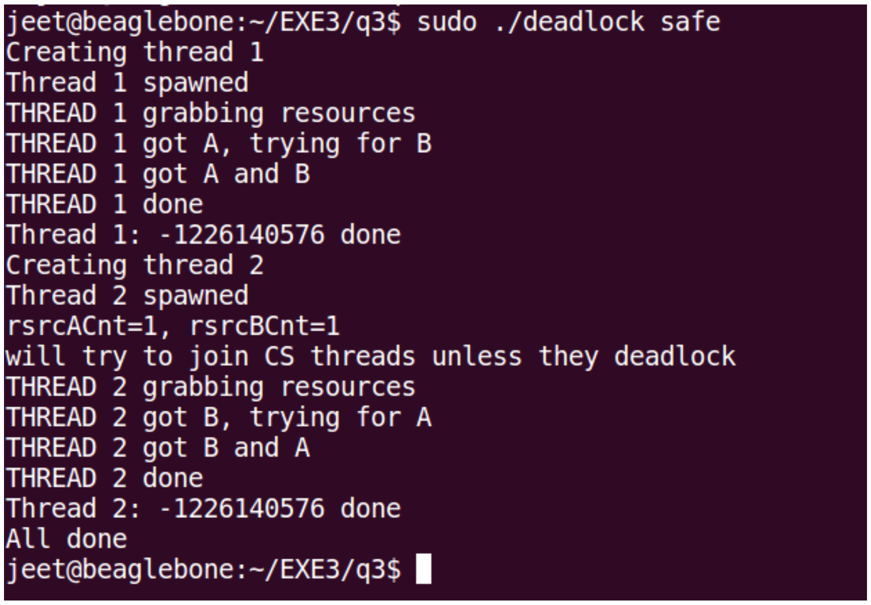
Question 3

In this question we implement the given code and analyze the occurrence of deadlock and unbounded priority inversion. We then modify the code to avoid such a scenario. Implementations of codes has been described below:



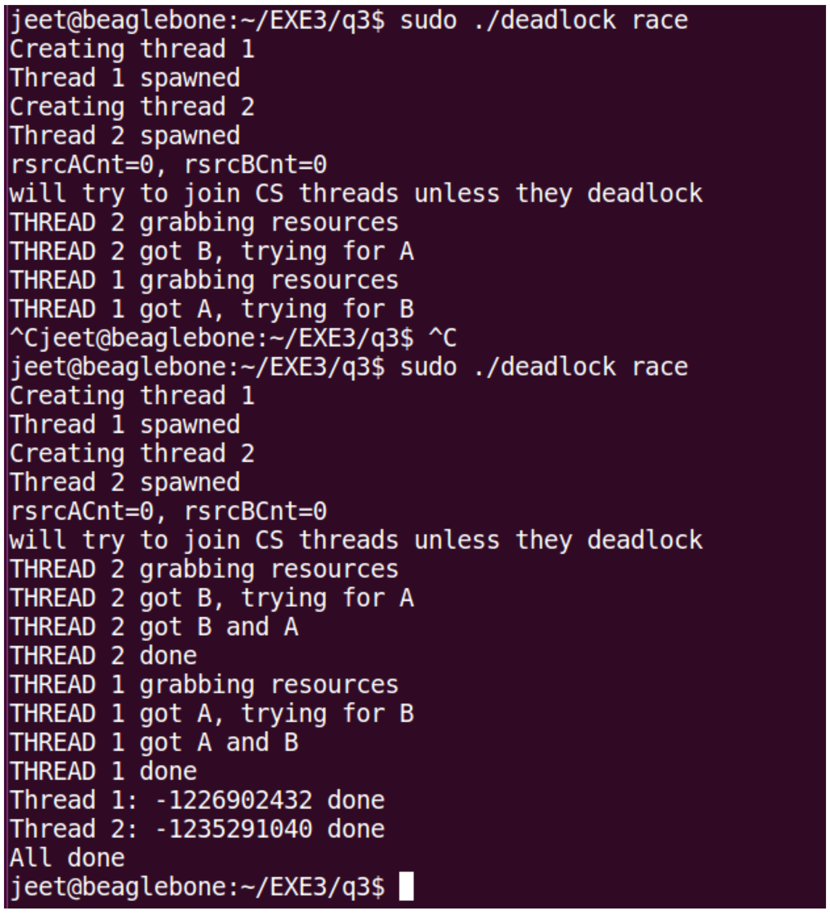
```
jeet@beaglebone:~/EXE3/q3$ sudo ./deadlock
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
rsrcACnt=0, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 grabbing resources
THREAD 2 got B, trying for A
THREAD 1 got A, trying for B
^Cjeet@beaglebone:~/EXE3/q3$ ^C
jeet@beaglebone:~/EXE3/q3$
```

The given code for deadlock has 2 threads. When no attributes like safe or race are mentioned then in such a condition the code ends in deadlock. The main thread creates thread-1 and then thread-2 and then it waits for first thread-1 to end and then waits for thread-2 to end. But thread-1 first grabs resource-A and goes into sleep for 1 second during which thread-2 runs and till then grabs resource-B and then it also goes into sleep for 1 second. Now thread-1 comes out of the sleep 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 the resource-A which is locked by thread-1. This leads to deadlocked condition of the program. Now the code provides provision of passing safe and race as arguments and we observe its functioning:



```
jeet@beaglebone:~/EXE3/q3$ sudo ./deadlock safe
Creating thread 1
Thread 1 spawned
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: -1226140576 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
THREAD 2 done
Thread 2: -1226140576 done
All done
jeet@beaglebone:~/EXE3/q3$
```

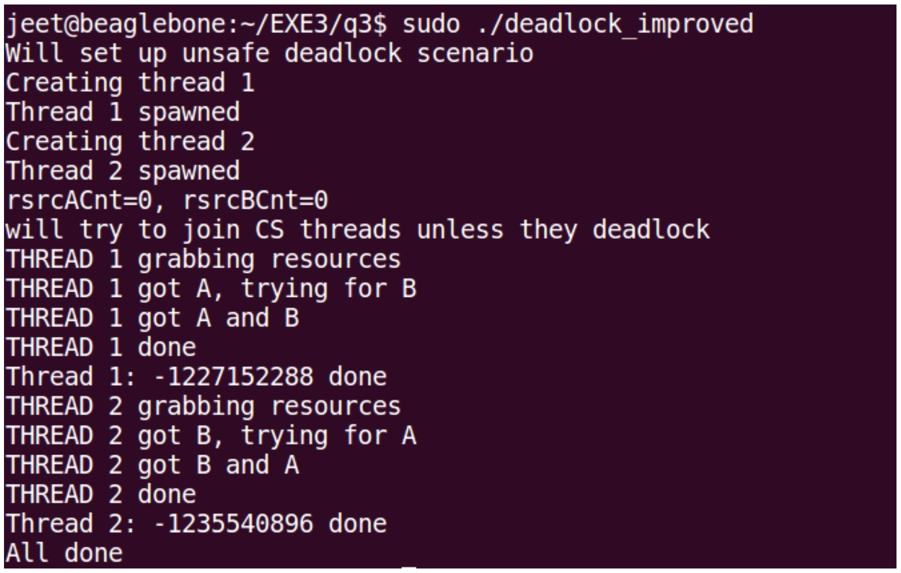
When the safe attribute is passed then the main first generates thread-1 and then waits for completion of its execution and then creates the thread-2 and then waits for completion of its execution. In this way both the threads never run simultaneously and hence deadlock is avoided.



```
jeet@beaglebone:~/EXE3/q3$ sudo ./deadlock race
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
rsrcACnt=0, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 2 got B, trying for A
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
^Cjeet@beaglebone:~/EXE3/q3$ ^C
jeet@beaglebone:~/EXE3/q3$ sudo ./deadlock race
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
rsrcACnt=0, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 2 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: -1226902432 done
Thread 2: -1235291040 done
All done
jeet@beaglebone:~/EXE3/q3$
```

When we pass the race attribute both threads 1 and 2 are generated one after the other and 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 in first execution we run into a deadlock condition. This is because although the wait statements are avoided and the resources are not block but since both threads run simultaneously the deadlock condition may arise. In the second execution we observe that there is no deadlock as no thread is waiting on the resource. Hence under the race conditions nothing can be certainly said about the occurrence of deadlocks.

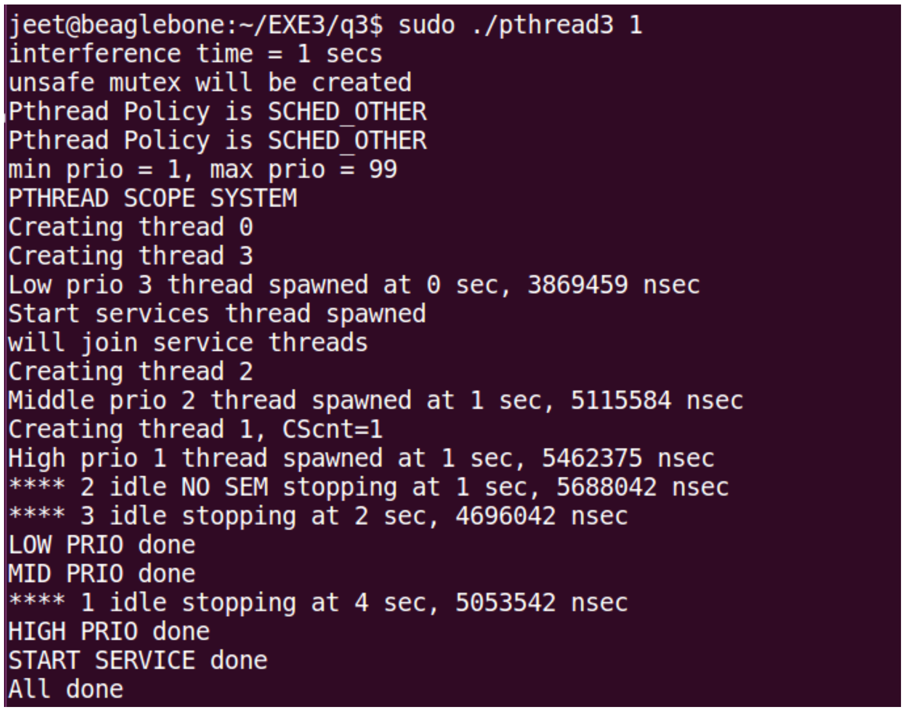
We now implement the Random Back Off method for preventing deadlocks. Here we delay one of the threads by some random time so that before the thread requests for a common resource the other thread has completed. The implementation has been described below:



```
jeet@beaglebone:~/EXE3/q3$ sudo ./deadlock_improved
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
rsrcACnt=0, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: -1227152288 done
THREAD 2 grabbing resources
THREAD 2 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
Thread 2: -1235540896 done
All done
```

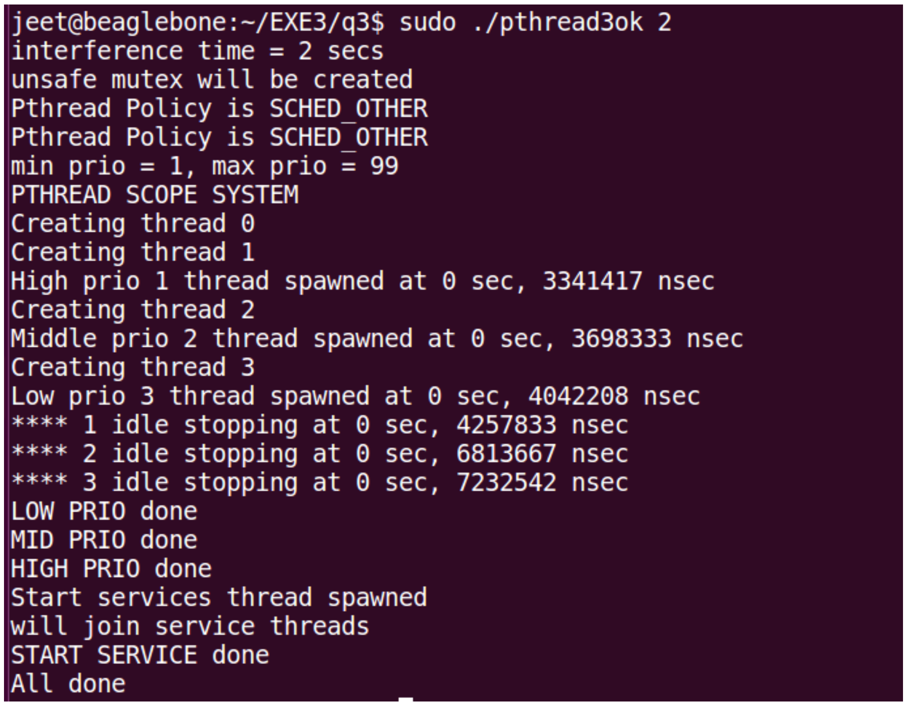
Here the deadlock code is modified such that when thread-1 grabs resource-A and goes to sleep then we make thread-2 backoff for some random time which is more than the time for which thread-1 sleeps. This ensures that there is enough time for thread-1 to acquire both the resources and complete its execution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

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



```
jeet@beaglebone:~/EXE3/q3$ 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 0 sec, 3869459 nsec
Start services thread spawned
will join service threads
Creating thread 2
Middle prio 2 thread spawned at 1 sec, 5115584 nsec
Creating thread 1, CScnt=1
High prio 1 thread spawned at 1 sec, 5462375 nsec
**** 2 idle NO SEM stopping at 1 sec, 5688042 nsec
**** 3 idle stopping at 2 sec, 4696042 nsec
LOW PRIO done
MID PRIO done
**** 1 idle stopping at 4 sec, 5053542 nsec
HIGH PRIO done
START SERVICE done
All done
```

In pthread3.c there are 3 services which are given priorities as low, medium and high. Out of the 3, the low and high priority services have to use mutex to perform their function while medium priority function is without any mutex. This is because service 1 and 3 share the same resource. So first the low priority service runs and it locks the resource and then the middle priority service which doesn't require mutex preempts the low priority service. Then the high priority service further preempts the middle priority service but as the resource is blocked by the low priority service, it cannot be completed. Hence in the end we see that lowest priority service finishes first and highest priority finishes at the last and this is the condition of priority inversion. Even though the higher priority service preempts it cannot execute due to the mutex lock on the required service. Also it has to wait till medium priority service executes this may be harmful if the system is hard real time.



```
jeet@beaglebone:~/EXE3/q3$ sudo ./pthread3ok 2
interference time = 2 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 0 sec, 3341417 nsec
Creating thread 2
Middle prio 2 thread spawned at 0 sec, 3698333 nsec
Creating thread 3
Low prio 3 thread spawned at 0 sec, 4042208 nsec
**** 1 idle stopping at 0 sec, 4257833 nsec
**** 2 idle stopping at 0 sec, 6813667 nsec
**** 3 idle stopping at 0 sec, 7232542 nsec
LOW PRIO done
MID PRIO done
HIGH PRIO done
Start services thread spawned
will join service threads
START SERVICE done
All done
```

In pthread3ok.c we create 3 services but there are no mutex hence there is no kind of resource blocking and hence the highest priority service finishes the first followed by the medium priority and then the lowest priority and hence there is no kind of priority inversion in this case. But such implementation is harful as the common resource which might be some data, might get corrupted as it is being written to and read from simultaneously by 2 threads hence such implementation is not advised. A solution to unbounded priority inversion is priority ceiling protocol and priority ceiling emulation protocol. In these implementations which the low priority is executing which having the locked resource, before being interrupted by medium priority, the priority of the task is set to such a high value that it cannot by preempted by the mid priority task. This way after completion of the previously low priority task the high priority task is first executed and mid priority is executed at the end. This becomes a much better approach when hard real time deadlines are involved.

PREEMPT_RT Patch as a Fix for priority inversion?

For unbounded inversion in LINUX, priority inheritance protocols can be used. The basic priority inheritance protocol bounded 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 one time and it also prevents deadlocks. PI-Futex already present in the Linux kernel can be used to implement priority inheritance, but it can still have inversion due to in-kernel spinlocks, semaphores or mutexes.

The PREEMPT-RT patch converts Linux into a fully preemptible kernel. The in-kernel locks can be pre-empted by using rtmutexes instead of the traditional locks. Even the critical sections guarded by rwlock or spinlocks can be preempted unless they are implemented using raw_spinlock_t. The interrupt handlers are also preemptible 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. As now the system level locks, critical sections and interrupts can be pre-empted, they can be used for priority inheritance to avoid priority inversion.

Based on inversion, we don't think we should switch to an RTOS and not use LINUX at all. Priority inversion is not only a Linux problem, it can happen even in an RTOS. With the PREEMPT RT patch Linux can be used for hard real time applications as well as soft real time applications. Though an RTOS would make more sense for an extreme case of hard real time application, it is not required to switch just based on inversion.

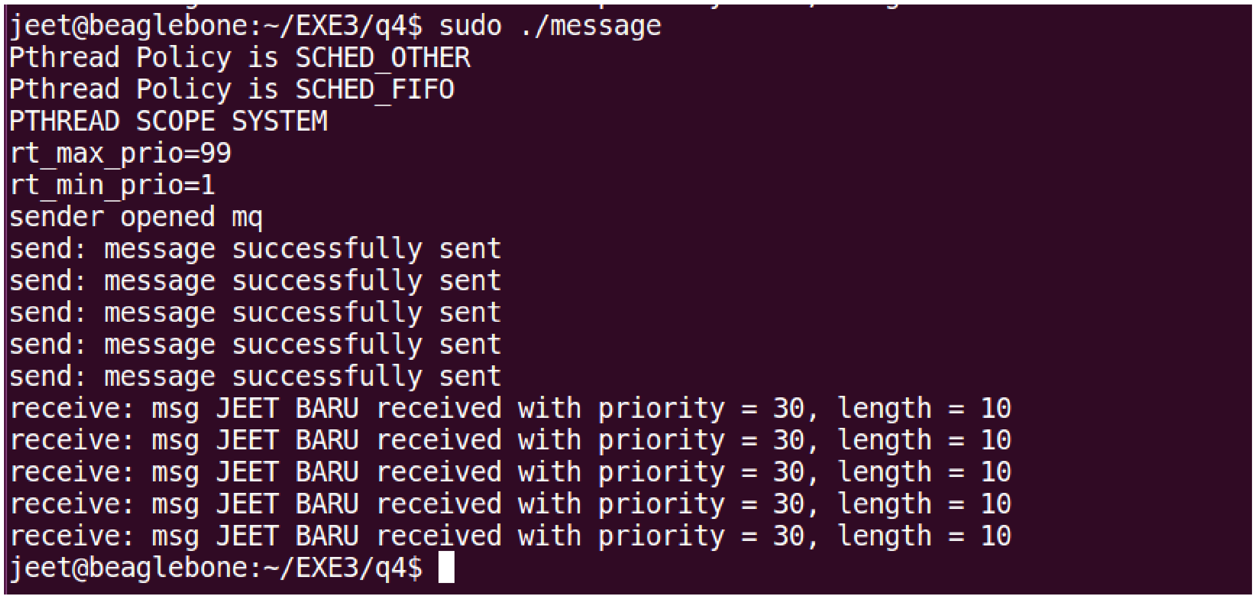
Question 4

Similarity between message queue and heap queue-

- Both the techniques ensure a safe way for the threads or processes to communicate with each other.
- Both are based on creating queues for storing the messages and the sender uses mq_send() to send the message to the queue and then the mq_receive is used read it back.
- Both the codes require to use the library mqueue.h to implement the queues and use all the functions in it to write and read.

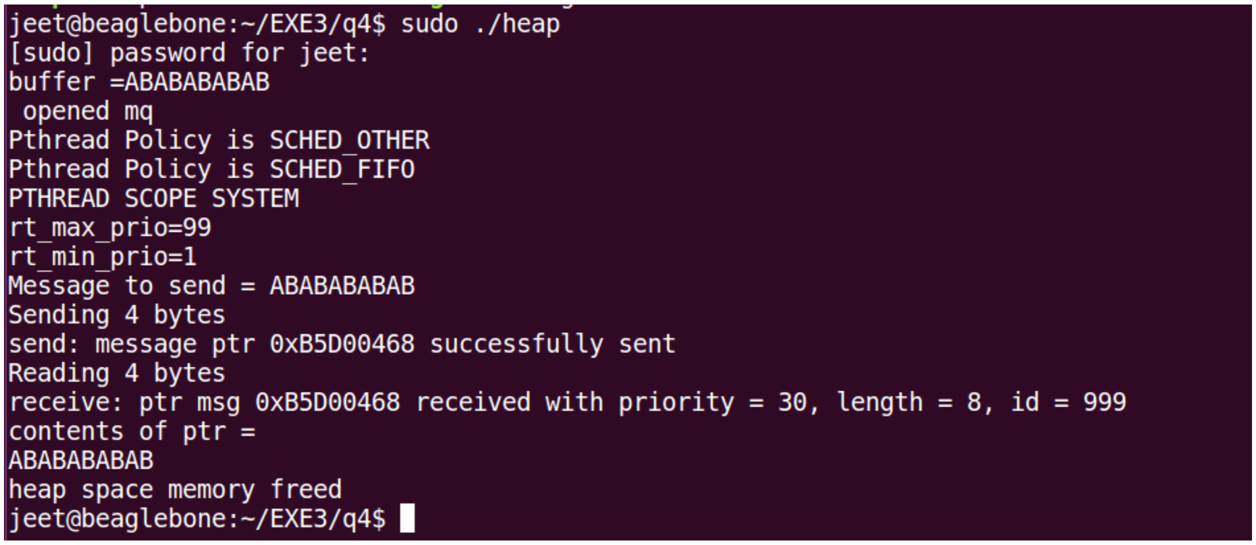
Differences between message queue and heap queue-

- Normal message queue uses message buffer in normal data memory, while in the heap queue the queues are implemented in the heap.
- 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.
- In message queue the efficiency is reduced because it requires a copy of the local buffer in the global message queue buffer which takes considerable CPU time and wastes memory due to double-buffering of the same data. This is not the case with heap queue because queue is created in the heap.
- For message queue the size of the message queue always remains the constant which is determined when the queue is opened, but in case of the heap queue, the sender allocate the buffer and the receiver de-allocate to avoid exhaustion of the associated buffer heap.
- 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.



```
jeet@beaglebone:~/EXE3/q4$ sudo ./message
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1
sender opened mq
send: message successfully sent
send: message successfully sent
send: message successfully sent
send: message successfully sent
send: message successfully sent
receive: msg JEET BARU received with priority = 30, length = 10
receive: msg JEET BARU received with priority = 30, length = 10
receive: msg JEET BARU received with priority = 30, length = 10
receive: msg JEET BARU received with priority = 30, length = 10
receive: msg JEET BARU received with priority = 30, length = 10
jeet@beaglebone:~/EXE3/q4$
```

Above is an example of normal message queue in which the sender and receiver both open up the queue. We specify various attributes like message queue size, length and priority while opening the queue. After the queue is opened we can send the message by using the mq_send() function and mq_receive() can be used receive the message.



```
jeet@beaglebone:~/EXE3/q4$ sudo ./heap
[sudo] password for jeet:
buffer =ABABABABAB
 opened mq
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1
Message to send = ABABABABAB
Sending 4 bytes
send: message ptr 0xB5D00468 successfully sent
Reading 4 bytes
receive: ptr msg 0xB5D00468 received with priority = 30, length = 8, id = 999
contents of ptr =
ABABABABAB
heap space memory freed
jeet@beaglebone:~/EXE3/q4$
```

Above code uses heap queue to transfer the message. Here again mq_open() is used to open up the queue but in the sender we specify a pointer to the heap so that the queue is opened up in the heap. And while using mq_send() we send the message to the heap allocating memory each time. And while reading the same we use mq_read() and de-allocate the memory.

Message queues solving unbounded priority inversion issue

As seen above priority inversion generally is a issue when we use mutexes and semaphores which do resource blocking in order to modify some global data. However removing the use of mutexes and semaphores completely is not feasible as we always require global data for communication between threads. Hence an alternative which can be used are message queues which give a good way to establish communication between threads and processes without locking of resources. However normal queues cannot always ensure that priority inversion will not take place. For ensuring that priority inversion doesn't take place POSIX message queue have a feature of queueing according to priority. It ensures that while dequeuing of the messages they are always dequeued such the highest priority message is dequeued first. So the user can wisely give the priority to the messages such that that priority inversion does not occur.

Question 5

Linux Watchdog daemon timer

The Linux kernel can reset the system if serious problems are detected. This can be implemented via special watchdog hardware, or via a slightly less reliable software-only watchdog inside the kernel. Usually a userspace daemon will notify the kernel watchdog driver via the "/dev/watchdog" special device file that userspace is still alive, at regular intervals. When such a notification occurs, the driver will usually tell the hardware watchdog that everything is in order, and that the watchdog should wait for yet another little while to reset the system. If userspace fails (RAM error, kernel bug, whatever), the notifications cease to occur, and the hardware watchdog will reset the system (causing a reboot) after the timeout occurs. To use the watchdog in linux we need a watchdog driver and the watchdog device file "/dev/watchdog". We can write to this file to give instructions to watchdog.

Under high system load watchdog might be swapped out of memory and may fail to make it back in in time. Under these circumstances the Linux kernel will reset the machine. To make sure you won't get unnecessary reboots make sure you have the variable realtime set to yes in the configuration file watchdog.conf. This adds real time support to watchdog: it will lock itself into memory and there should be no problem even under the highest of loads.

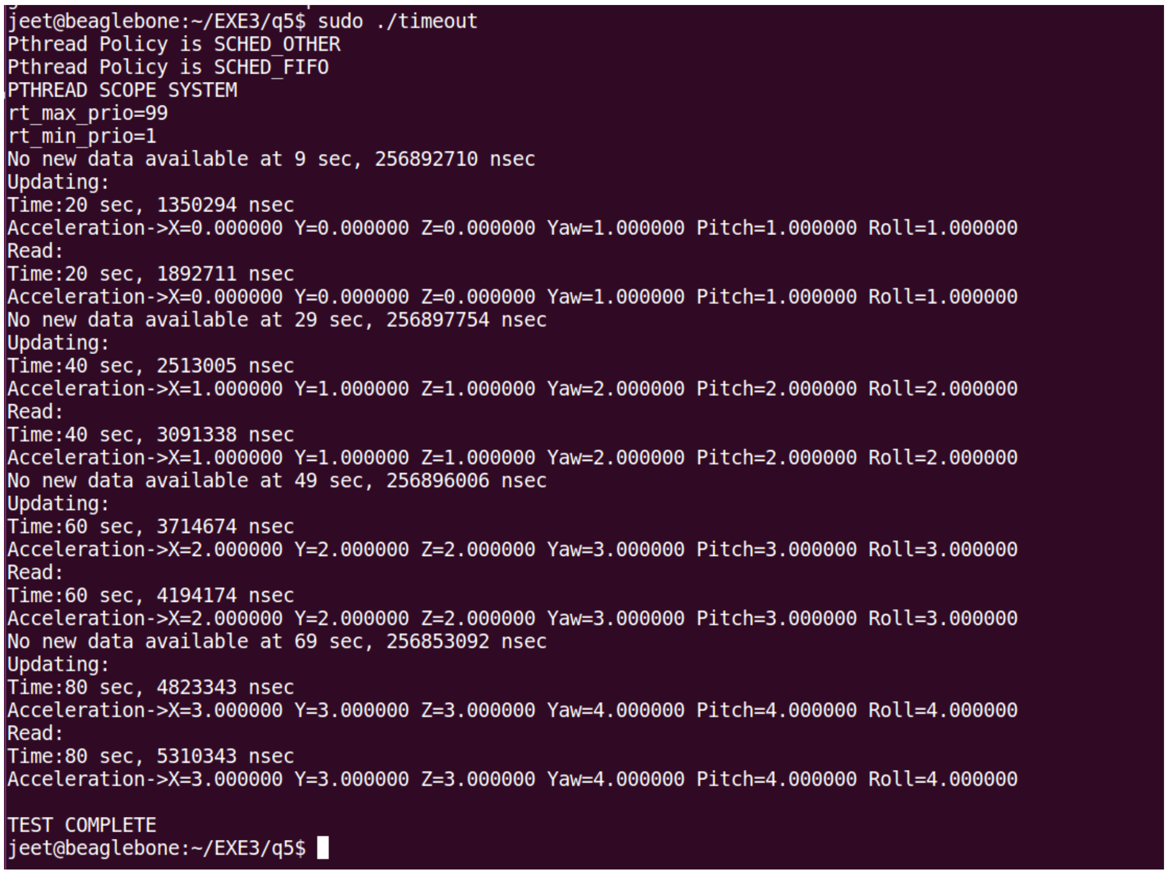
API calls can be used in the to give various type of instructions to the watchdog timer. The normal API usually always working, where the watchdog activates as soon as /dev/watchdog is opened and will reboot unless the watchdog is pinged within a certain time, this time is called the timeout or margin. The ioctl API can be used to write various instructions to the watchdog timer and it can be used to configure timeouts as well. It is possible to set timeouts as well as read the timeouts using the ioctl API. For some drivers it is possible to modify the watchdog timeout on the fly with the SETTIMEOUT ioctl, those drivers have the WDIOF_SETTIMEOUT flag set in their option field. The argument is an integer representing the timeout in seconds. The driver returns the real timeout used in the same variable, and this time out might differ from the requested one due to limitation of the hardware. Some watchdog timers can be set to have a trigger to go off before the actual time they will reset the system using the SETPRETIMEOUT ioctl. Hence using these APIs we can adjust the timeout of the watchdog timer which can help us to resolve the deadlocks.

Watchdog for resolving the deadlocks

Deadlock is a condition in which a service-1 locks a resource which another service-2 requires and service-2 has locked a resource which service-1 requires. Hence both the services are blocked indefinitely and this causes the system to lock and it causes a system halt. Hence the deadlocks are very critical and it is very important to somehow resolve them.

Generally, the supervisor service is required to reset the watchdog timer continuously to ensure that it doesn't time out. However, if the supervisor service is deadlocked then it is not able to reset the watchdog timer and so the system undergoes a hardware reset. This causes the firmware to reboot the system in hopes that the supervisor deadlock will not evolve again. As discussed above it is possible to change the timeouts of the watchdog timer by using the ioctl APIs. The default timeout of the watchdog timer will be 60 seconds; we can change the timeouts as per our need.

Now we modify our code from question 2 using pthread_mutex_timedlock(). We use this function to wait and hold the mutex for a specific amount of time and then release the lock. The implementation is as shown below:



```
jeet@beaglebone:~/EXE3/q5$ sudo ./timeout
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1
No new data available at 9 sec, 256892710 nsec
Updating:
Time:20 sec, 1350294 nsec
Acceleration->X=0.000000 Y=0.000000 Z=0.000000 Yaw=1.000000 Pitch=1.000000 Roll=1.000000
Read:
Time:20 sec, 1892711 nsec
Acceleration->X=0.000000 Y=0.000000 Z=0.000000 Yaw=1.000000 Pitch=1.000000 Roll=1.000000
No new data available at 29 sec, 256897754 nsec
Updating:
Time:40 sec, 2513005 nsec
Acceleration->X=1.000000 Y=1.000000 Z=1.000000 Yaw=2.000000 Pitch=2.000000 Roll=2.000000
Read:
Time:40 sec, 3091338 nsec
Acceleration->X=1.000000 Y=1.000000 Z=1.000000 Yaw=2.000000 Pitch=2.000000 Roll=2.000000
No new data available at 49 sec, 256896006 nsec
Updating:
Time:60 sec, 3714674 nsec
Acceleration->X=2.000000 Y=2.000000 Z=2.000000 Yaw=3.000000 Pitch=3.000000 Roll=3.000000
Read:
Time:60 sec, 4194174 nsec
Acceleration->X=2.000000 Y=2.000000 Z=2.000000 Yaw=3.000000 Pitch=3.000000 Roll=3.000000
No new data available at 69 sec, 256853092 nsec
Updating:
Time:80 sec, 4823343 nsec
Acceleration->X=3.000000 Y=3.000000 Z=3.000000 Yaw=4.000000 Pitch=4.000000 Roll=4.000000
Read:
Time:80 sec, 5310343 nsec
Acceleration->X=3.000000 Y=3.000000 Z=3.000000 Yaw=4.000000 Pitch=4.000000 Roll=4.000000

TEST COMPLETE
jeet@beaglebone:~/EXE3/q5$
```

In this case 2 threads exist, one which updates the data locks the mutex after 20 seconds and the second thread tries to obtain the mutex to read the data. The wait in the second thread is timed out after 10 seconds and it prints a message if no change in data is made, and goes back to normal wait state. Hence the pthread_mutex_timedlock has been implemented.