3.

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

ANSWER

Deadlock

Deadlock is a condition that takes place when two or more threads are looking for more than one shared piece of memory. For example: Let us assume a situation wherein one thread takes shared memory A and another thread takes shared memory B. After a while, when the respective memory access is still happening, there arises a condition where the first thread asks for shared memory B and the second thread asks for shared memory A. In such a case, execution of each thread will stop indefinitely. A solution to this issue is Random Back-off, which is to give up the resource that a thread owns, when it tries and fails to get the other resource. After this, the thread waits for a pseudo-random amount of time and tries to grab the resource again. In that wait period, it is expected that the other conflicting thread will get access to both pieces of memory and finish executing its task with the shared memory. This has been implemented on the deadlock code given in the problem.

Priority Inversion

Priority Inversion is an effect of having multiple priority level tasks/services and shared resources. When a low priority task and a high priority task share the same data, the former might end up blocking the latter. Essentially, while the low priority task will have the shared memory and would be executing on it, the high priority task would need to wait for it to finish. During this time, though, a medium priority task could pre-empt the low priority task and thus force the high priority task to wait for many lower priority tasks to finish execution, delaying its access to the shared memory more and more. This is the process by which Priority Inversion takes place. It is demonstrated in the code given and can be seen executing.

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.

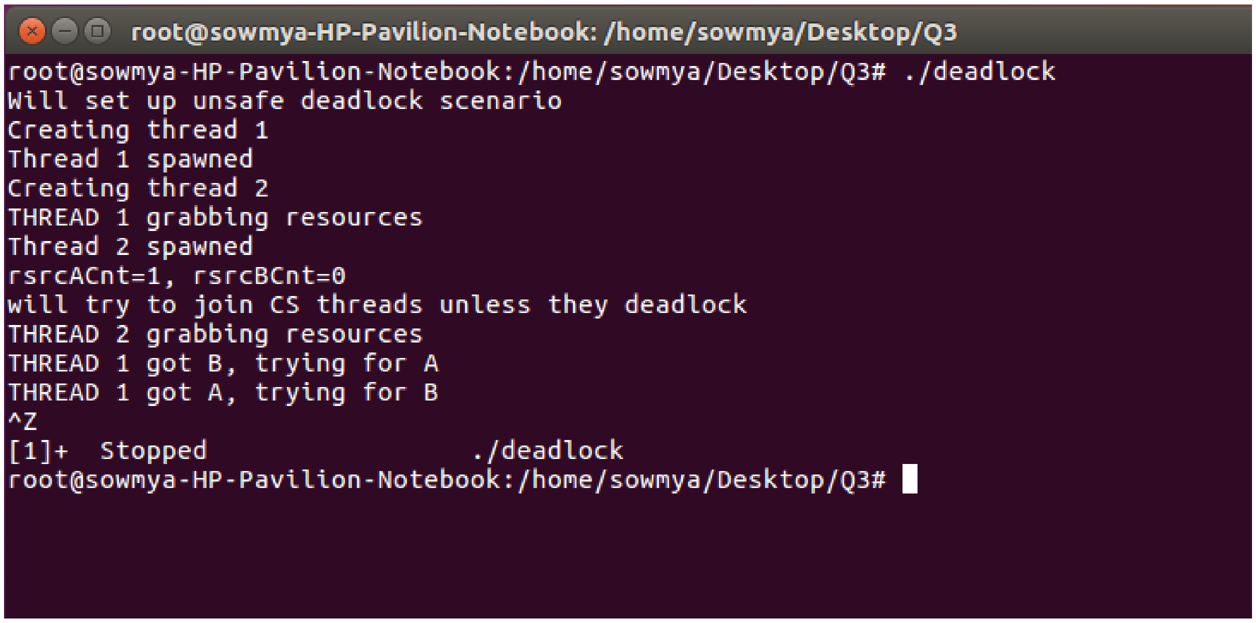
Implementation and correction of code

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.

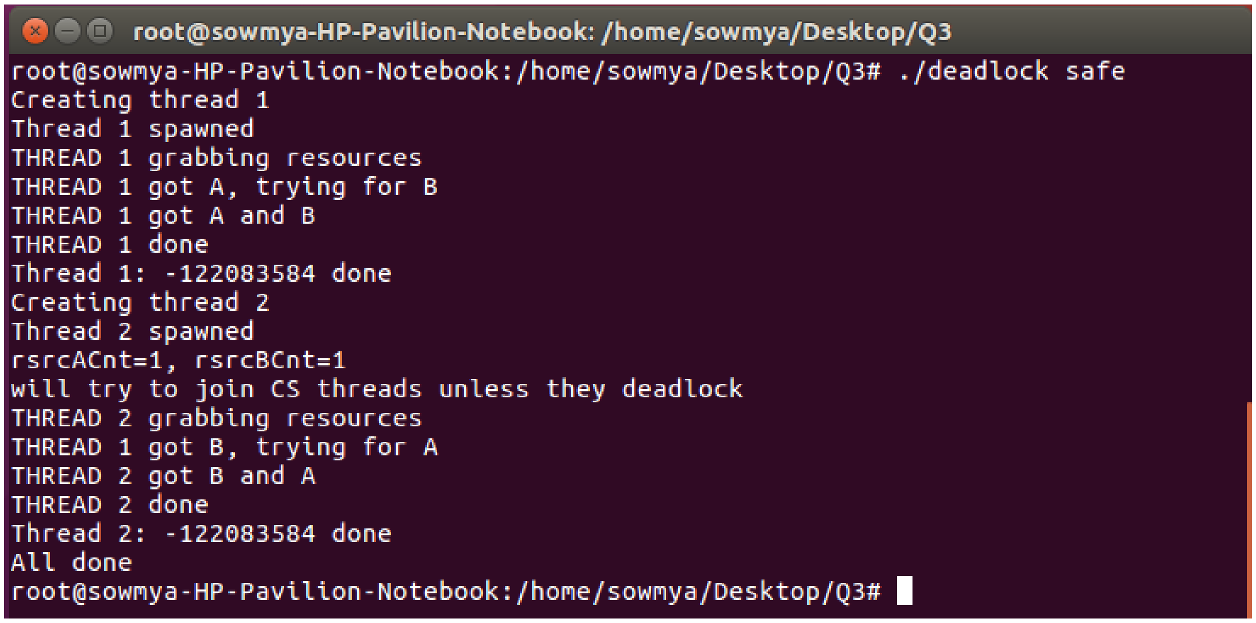


```
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Desktop/Q3# ./deadlock
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
THREAD 1 grabbing resources
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
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 thread-2 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.

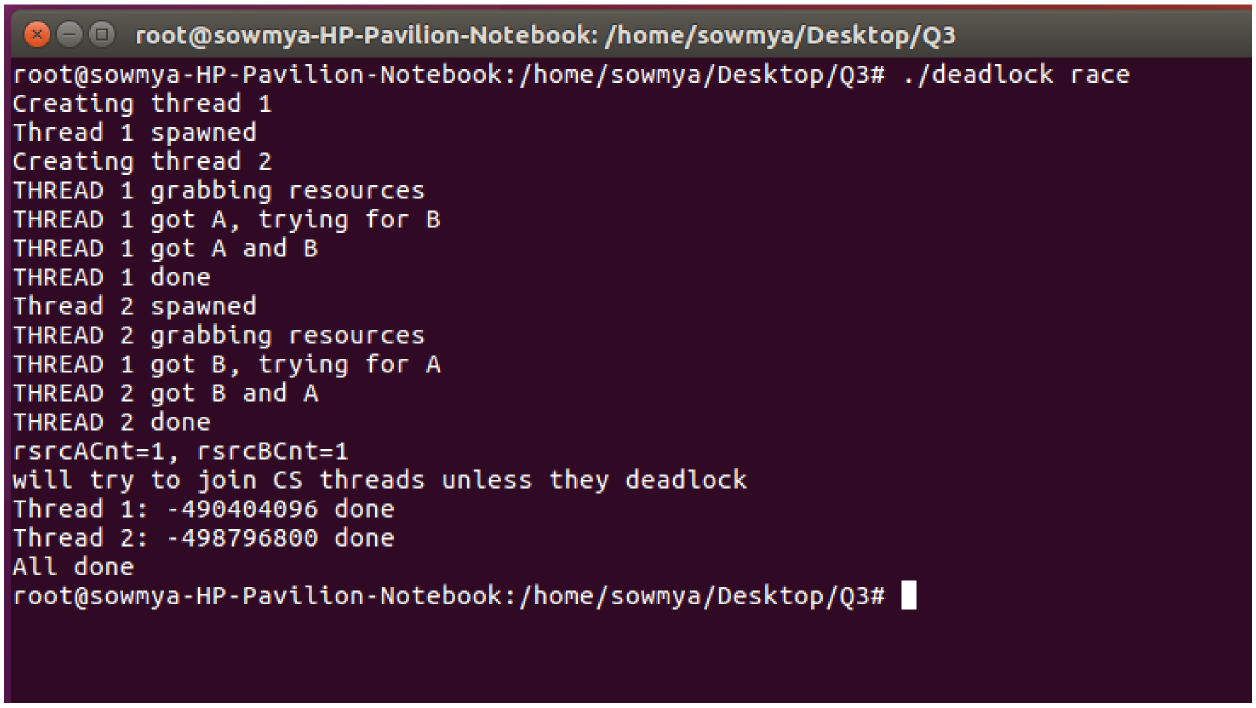
deadlock_safe.c



```
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.

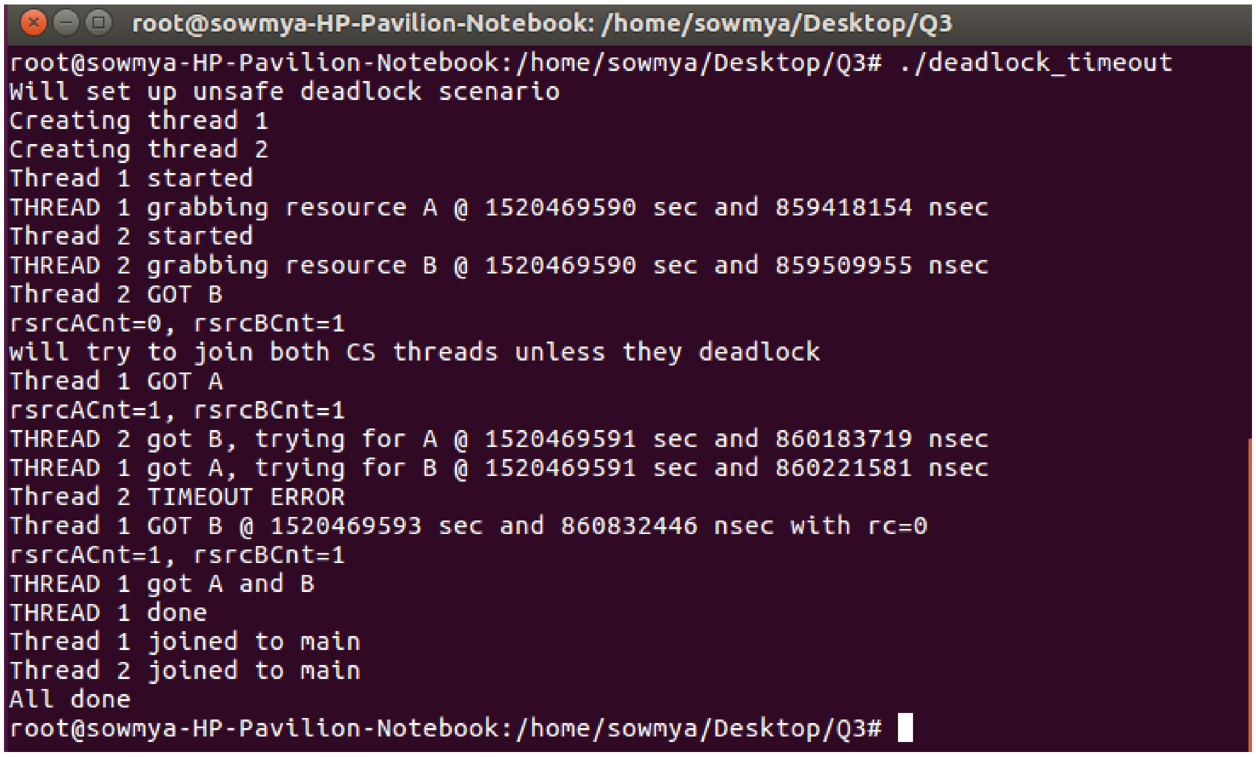
deadlock_race.c



```
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

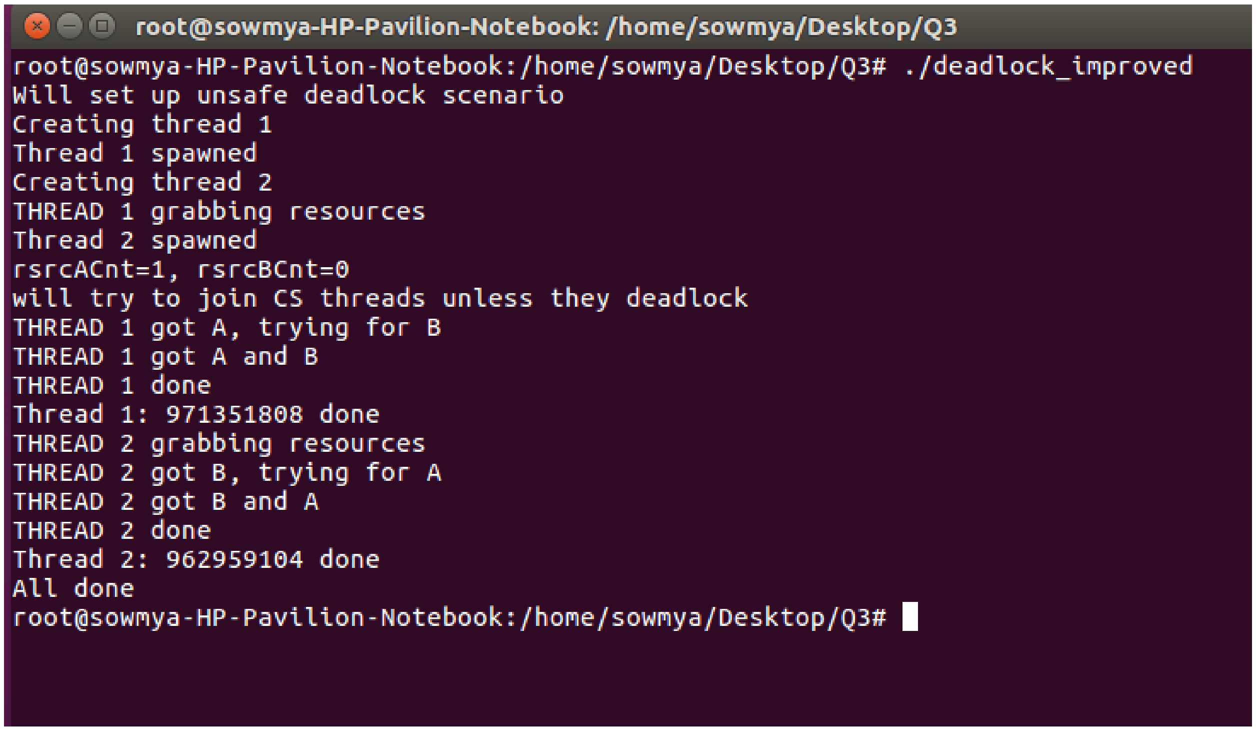


```
root@sowmya-HP-Pavilion-Notebook: /home/sowmya/Desktop/Q3
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Desktop/Q3# ./deadlock_timeout
Will set up unsafe deadlock scenario
Creating thread 1
Creating thread 2
Thread 1 started
THREAD 1 grabbing resource A @ 1520469590 sec and 859418154 nsec
Thread 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 resource-B. 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

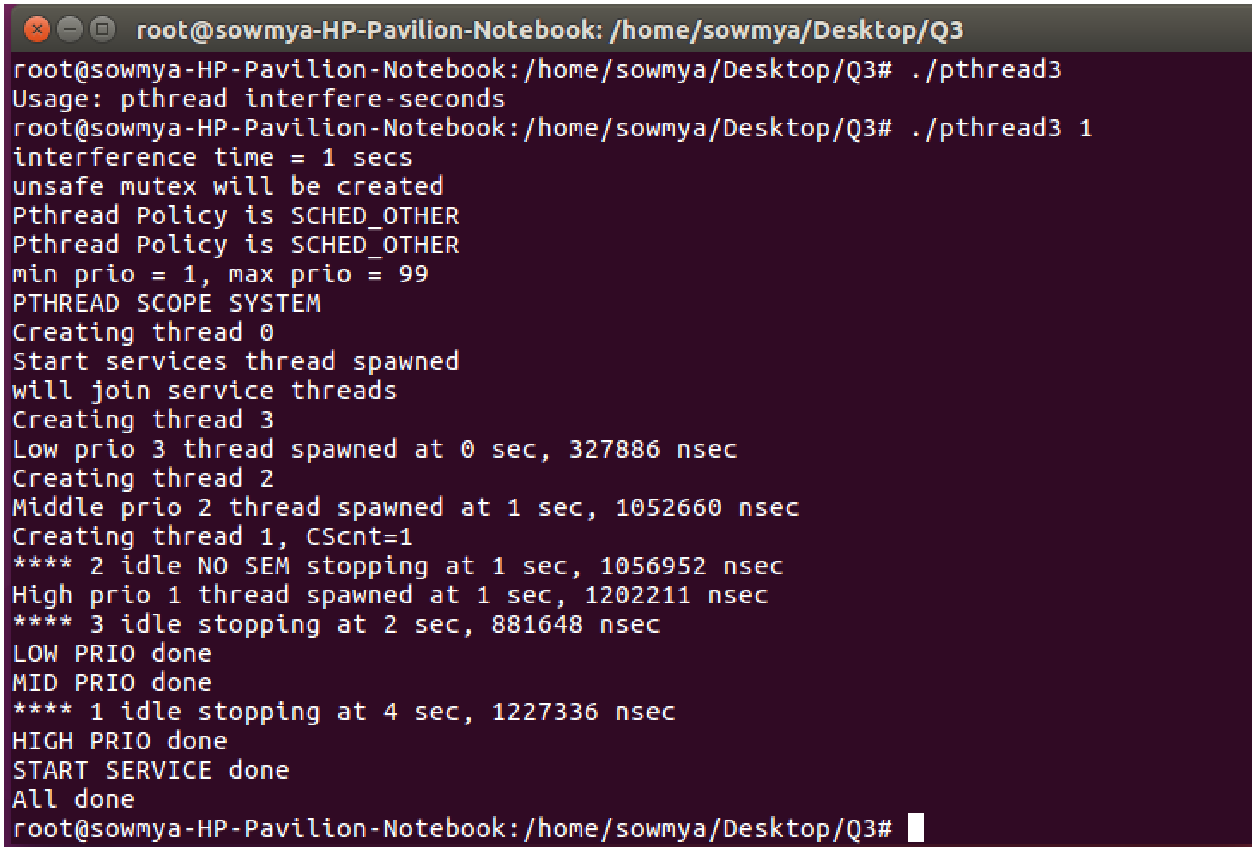


```
root@sowmya-HP-Pavilion-Notebook: /home/sowmya/Desktop/Q3
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Desktop/Q3# ./deadlock_improved
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
THREAD 1 grabbing resources
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=0
will 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, thread-2 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.

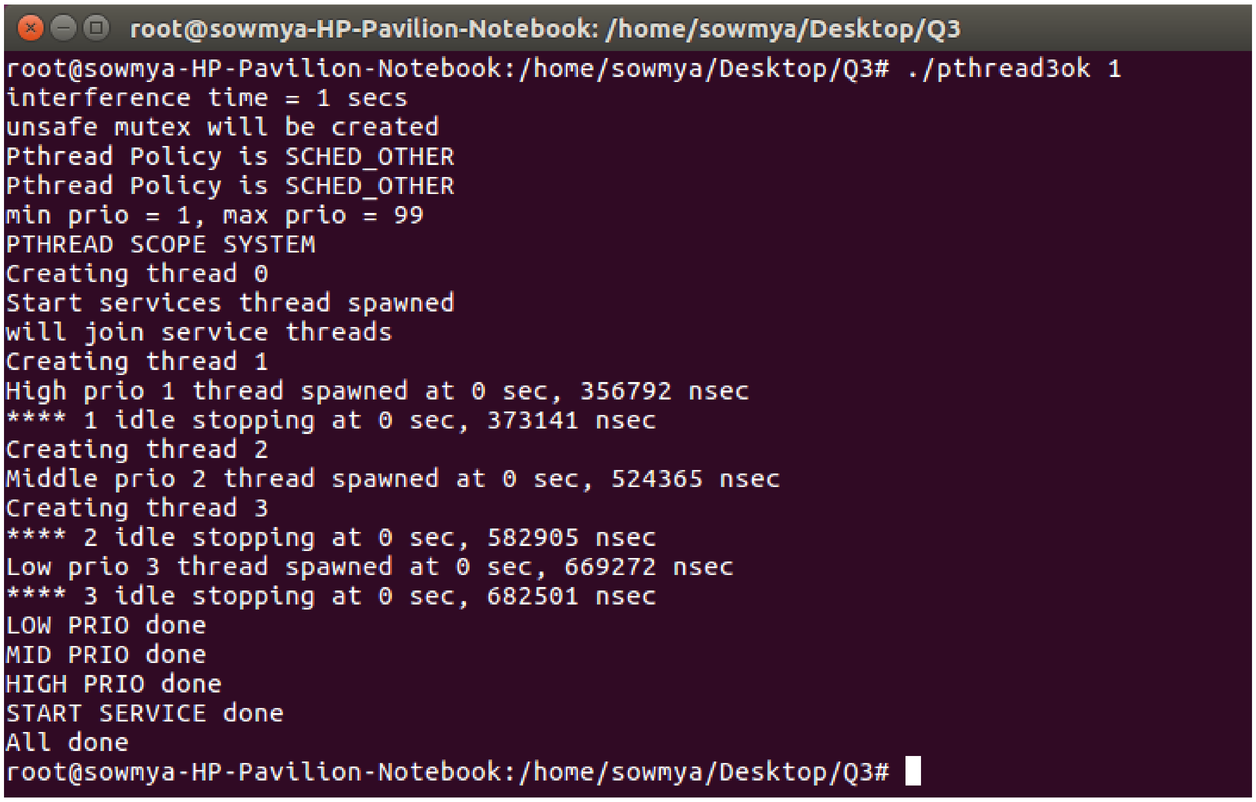
pthread3.c



```
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# ./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.