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 backoff 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?

**Demonstration and description of deadlock with threads:**

In real-time embedded systems, deadlock is a situation that occurs when multiple threads are waiting on resources that are being held by one another. This is typically done in a circular fashion when presented on a graph. Unbounded priority inversion occurs when a higher priority thread is unable to execute due to a resource being held by a lower priority thread. In addition to this, a medium priority thread may interfere lower priority thread for an indeterminate amount of time and as a result prevents the higher priority thread from running.

In the instance of deadlock.c, two threads are spawned by the code: THREAD 1 and THREAD 2. When THREAD 1 is spawned it locks (or grabs) resource A which is a pthread mutex. THREAD 2 is then spawned and in a similar manner, locks resource B. Once THREAD 1 has finished executing some code, it now requires resource B however this is locked by THREAD 2. After this, THREAD 2 now requires resource A. Both threads require both resources at the same time, however each thread has one of the resources locked which results in deadlock.

Description of RT\_PREEMPT\_PATCH and assessment of whether Linux can be made real-time safe: The Linux kernel by itself is not real time capable. But, with the addition of the RT\_PREEMPT\_PATCH it gains real-time capabilities. The main aim of introducing this patch is to make the existing Linux kernel preemptible. The RT\_PREEMPT\_PATCH provides features like preemptible critical sections, interrupt handlers, priority inherited in-kernel spinlocks and semaphores to make the kernel preemptible and provide a much better response time, thus making Linux + RT\_PREEMPT\_PATCH closer to resemble the RTOS. With this patch, we can assign priorities even to kernel tasks such that user space threads can preempt them. These features could help solve the unbounded priority inversion issue. However, due to a small amount of latency in the Linux kernel the unbounded priority inversion cannot be fully solved. Locking mechanisms like spinlocks are implemented as mutexes and semaphores which help to preserve the critical sections against data corruption.

I think that the decision to switch from a Linux OS to an RTOS depends upon a variety of factors for an application. It depends if the system requirement is strictly hard real-time or if the system can afford to miss a few deadlines and still work safely. Predictability of the system is often decided by the kind of scheduling algorithm the system follows. Safety is one more factor which is crucial in deciding the requirements of the system. In hard real-time systems, safety is often the most important factor to consider. On the other hand, safety can be overlooked to some extent in a soft real time system. To sum up, I’d say if an application demands features which resemble to a hard-real time system then switching to an RTOS makes more sense. However, if the requirements of the system to be designed is not too strict, then a RT Linux can be a good platform to build that application.