1.

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

ANSWER

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

- For tasks in priority-driven pre-emptive real-time systems, when semaphores are used for synchronization shared memory access, blocking issues can arise, with lower priority tasks blocking a task of high priority for an indefinite period of time. In order to solve this issue, priority inheritance techniques should be used.
- It can be seen from the understanding of the paper that a job can be blocked at most once per semaphore it uses. [Theorem 6] Thus, the higher priority task can be blocked for a maximum of 'm' times if there are 'm' semaphores. This point is key, since the blocking time for each semaphore must be bounded – assuming no job has infinite execution. This results in a job that has a finite upper bound on its blocking time.
- Priority inheritance lets the lower priority task, which has locked the semaphore, inherit the priority of the higher priority task such that it runs as the higher priority task. Thus, one task's priority is inherited by another. While this does solve the problem of unbounded blocking, it still has issues in that it cannot prevent deadlocks. Also, the unbounded blocking period can have significance due to chain of blocking by other tasks.
- In order to solve the issues of deadlock and chained blocking, and to bound the blocking period, the priority ceiling protocol is used. Each task has a basic priority inheritance associated with it. Priority ceiling protocol assigns a priority ceiling to each semaphore. This results in an inherited higher priority task blocking a high priority task while sharing multiple semaphores such that the inherited task gets all the required semaphores. This effectively avoids deadlocks and chained blocking. Thus, a higher priority task can be blocked by a lower priority task for at most one critical section of all critical sections of the lower priority task. [Lemma 11]
- Another important note on the priority ceiling protocol is that there isn't any transitive blocking with this protocol. Thus, a low priority task will immediately be pushed to the highest priority of all tasks it can block for any given semaphore. [Lemma 9]

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

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

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

Futexes as a solution for Unbounded Priority Inversion

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

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