WIKIPEDIA
The Free Encyclopedia

Priority inversion

In computer science, **priority inversion** is a scenario in scheduling in which a high priority task is indirectly superseded by a lower priority task effectively inverting the assigned priorities of the tasks. This violates the priority model that high-priority tasks ca[illegible]ly be prevented from running by higher-priority tasks. Inversion occurs when there is a resource contention with a low-priority task that is then preempted by a medium-priority task.

Formulation

Consider two tasks H and L, of high and low priority respectively, either of which can acquire exclusive use of a shared resource R. If H attempts to acquire R after L has acquired it, then H becomes blocked until L relinquishes the resource. Sharing an exclusive-use resource (R in this case) in a well-designed system typically involves L relinquishing R promptly so that H (a higher priority task) does not stay blocked for excessive periods of time. Despite good design, however, it is possible that a third task M of medium priority becomes runnable during L's use of R. At this point, M being higher in priority than L, preempts L (since M does not depend on R), causing L to not be able to relinquish R promptly, in turn causing H—the highest priority process—to be unable to run (that is, H suffers unexpected blockage indirectly caused by lower priority tasks like M).

Consequences

In some cases, priority inversion can occur without causing immediate harm—the delayed execution of the high-priority task goes unnoticed, and eventually, the low-priority task releases the shared resource. However, there are also many situations in which priority inversion can cause serious problems. If the high-priority task is left starved of the resources, it might lead to a system malfunction or the triggering of pre-defined corrective measures, such as a watchdog timer resetting the entire system. The trouble experienced by the Mars Pathfinder lander in 1997[1][2] is a classic example of problems caused by priority inversion in realtime systems.

Priority inversion can also reduce the perceived performance of the system. Low-priority tasks usually have a low priority because it is not important for them to finish promptly (for example, they might be a batch job or another non-interactive activity). Similarly, a high-priority task has a high priority because it is more likely to be subject to strict time constraints—it may be providing data to an interactive user, or acting subject to real-time response guarantees. Because priority inversion results in the execution of a lower-priority task blocking the high-priority task, it can lead to reduced system responsiveness or even the violation of response time guarantees.

A similar problem called deadline interchange can occur within earliest deadline first scheduling (EDF).

Solutions

The existence of this problem has been known since the 1970s. Lampson and Redell [3] published one of the first papers to point out the priority inversion problem. Systems such as the UNIX kernel were already addressing the problem with the splx() primitive. There is no foolproof method to predict the situation. There are however many existing solutions, of which the most common ones are:

Disabling all interrupts to protect critical sections

When disabling interrupts is used to prevent priority inversion, there are only two priorities: *preemptible*, and *interrupts disabled.* With no third priority, inversion is impossible. Since there's only one piece of lock data (the interrupt-enable bit), misordering locking is impossible, and so deadlocks cannot occur. Since the critical regions always run to completion, hangs do not occur. Note that this only works if all interrupts are disabled. If only a particular hardware device's interrupt is disabled, priority inversion is reintroduced by the hardware's prioritization of interrupts. In early versions of UNIX, a series of primitives named splx(0) ... splx(7) disabled all interrupts up through the given priority. By properly choosing the highest priority of any interrupt that ever entered the critical section, the priority inversion problem could be solved without locking out all of the interrupts. Ceilings were assigned in rate-monotonic order, i.e. the slower devices had lower priorities.

In multiple CPU systems, a simple variation, "single shared-flag locking" is used. This scheme provides a single flag in shared memory that is used by all CPUs to lock all inter-processor critical sections with a busy-wait. Interprocessor communications are expensive and slow on most multiple CPU systems. Therefore, most such systems are designed to minimize shared resources. As a result, this scheme actually works well on many practical systems. These methods are widely used in simple embedded systems, where they are prized for their reliability, simplicity and low resource use. These schemes also require clever programming to keep the critical sections very brief. Many software engineers consider them impractical in general-purpose computers.

Priority ceiling protocol

With priority ceiling protocol, the shared mutex process (that runs the operating system code) has a characteristic (high) priority of its own, which is assigned to the task of locking the mutex. This works well, provided the other high-priority task(s) that tries to access the mutex does not have a priority higher than the ceiling priority.

Priority inheritance

Under the policy of priority inheritance, whenever a high-priority task has to wait for some resource shared with an executing low-priority task, the low-priority task is temporarily assigned the priority of the highest waiting priority task for the duration of its own use of the shared resource, thus keeping medium priority tasks from pre-empting the (originally) low priority task, and thereby affecting the waiting high priority task as well. Once the resource is released, the low-priority task continues at its original priority level.

Random boosting

Ready tasks holding locks are randomly boosted in priority until they exit the critical section. This solution is used in Microsoft Windows.[4]

Avoid blocking

Because priority inversion involves a low-priority task blocking a high-priority task, one way to avoid priority inversion is to avoid blocking, for example by using non-blocking algorithms such as read-copy-update.

See also

- Nice (Unix)
- Non-blocking synchronization

- Pre-emptive multitasking

References

1. Glenn Reeves, *What Really Happened on Mars* (https://www.cs.unc.edu/~anderson/teach/comp790/papers/mars_pathfinder_long_version.html), JPL Pathfinder team, retrieved 2019-01-04
2. *Explanation of priority inversion problem experienced by Mars Pathfinder* (https://www3.nd.edu/~cpoellab/teaching/cse40463/slides6.pdf) (PDF), retrieved 2019-01-04
3. Lampson, B; Redell, D. (June 1980). "Experience with processes and monitors in MESA". *Communications of the ACM*. **23** (2): 105–117. CiteSeerX 10.1.1.46.7240 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.46.7240). doi:10.1145/358818.358824 (https://doi.org/10.1145%2F358818.358824). S2CID 1594544 (https://api.semanticscholar.org/CorpusID:1594544).
4. Priority Inversion (https://msdn.microsoft.com/en-us/library/windows/desktop/ms684831(v=vs.85).aspx) on MSDN

External links

- Description from FOLDOC (http://foldoc.org/priority+inversion)
- Citations from CiteSeer (http://citeseer.org/cs?q=priority+inversion)
- IEEE Priority Inheritance Paper by Sha, Rajkumar, Lehoczky (http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=626613)
- Introduction to Priority Inversion by Michael Barr (https://www.embedded.com/electronics-blogs/beginner-s-corner/4023947/Introduction-to-Priority-Inversion)