Introduction to Priority Inversion

Michael Barr

4/1/2002 10:06 AM EST

Introduction to Priority Inversion

When tasks share resources, as they often do, strange things can and will happen. Priority inversions can be particularly difficult to anticipate. Here's an introduction to priority inversions and a pair of techniques you can use to avoid them.

Most commercial real-time operating systems (RTOSes) employ a priority-based preemptive scheduler. These systems assign each task a unique priority level. The scheduler ensures that *of those tasks that are ready to run*, the one with the highest priority is always the task that is actually running. To meet this goal, the scheduler may preempt a lower-priority task in mid-execution.

Because tasks share resources, events outside the scheduler's control can prevent the highest priority ready task from running when it should. If this happens, a critical deadline could be missed, causing the system to fail. *Priority inversion* is the term for a scenario in which the highest-priority ready task fails to run when it should.

Resource sharing

Tasks need to share resources to communicate and process data. This aspect of multi-threaded programming is not specific to real-time or embedded systems.

Any time two tasks share a resource, such as a memory buffer, in a system that employs a priority-based scheduler, one of them will usually have a higher priority. The higher-priority task expects to be run as soon as it is ready. However, if the lower-priority task is using their shared resource when the higher-priority task becomes ready to run, the higher-priority task must wait for the lower-priority task to finish with it. We say that the higher-priority task is *pending* on the resource. If the higher-priority task has a critical deadline that it must meet, the worst-case "lockout time" for all of its shared resources must be calculated and taken into account in the design. If the cumulative lockout times are too long, the resource-sharing scheme must be redesigned.

Since worst-case delays resulting from the sharing of resources can be calculated at design time, the only way they can affect the performance of the system is if no one properly accounts for them.

Priority inversions

The real trouble arises at run-time, when a medium-priority task preempts a lower-priority task using a shared resource on which the higher-priority task is pending. If the higher-priority task is otherwise ready to run, but a medium-priority task is currently running instead, a priority inversion is said to occur.

Figure 1 Priority inversion timeline

This dangerous sequence of events is illustrated in Figure 1. Low-priority Task L and high-priority Task H share a resource. Shortly after Task L takes the resource, Task H becomes ready to run. However, Task H must wait for Task L to finish with the resource, so it pends. Before Task L finishes with the resource, Task M becomes ready to run, preempting Task L. While Task M (and perhaps additional intermediate-priority tasks) runs, Task H, the highest-priority task in the system, remains in a pending state.

Many priority inversions are innocuous or, at most, briefly delay a task that should run right away. But from time to time a system-critical priority inversion takes place. Such an event occurred on the Mars Pathfinder mission in July 1997. The Pathfinder mission is best known for the little rover that took high-resolution color pictures of the Martian surface and relayed them back to Earth.

The problem was not in the landing software, but in the mission software run on the Martian surface. In the spacecraft, various devices communicated over a MIL-STD-1553 data bus. Activity on this bus was managed by a pair of high-priority tasks. One of the bus manager tasks communicated through a pipe with a low-priority meteorological science task.

On Earth, the software mostly ran without incident. On Mars, however, a problem developed that was serious enough to trigger a series of software resets during the mission. The sequence of events leading to each reset began when the low-priority science task was preempted by a couple of medium-priority tasks while it held a mutex related to the pipe. While the low-priority task was preempted, the high-priority bus distribution manager tried to send more data to it over the same pipe. Because the mutex was still held by the science task, the bus distribution manager was made to wait. Shortly thereafter, the other bus scheduler became active. It noticed that the distribution manager hadn't completed its work for that bus cycle and forced a system reset.

This problem was not caused by a mistake in the operating system, such as an incorrectly implemented semaphore, or in the application. Instead, the software exhibited behavior that is a known "feature" of semaphores and intertask communication. In fact, the RTOS used on Pathfinder featured an optional priority-inversion workaround; the scientists at JPL simply hadn't been aware of that option. Fortunately, they were able to recreate the problem on Earth, remotely enable the workaround, and complete the mission successfully.

Workarounds

Research on priority inversion has yielded two solutions. The first is called *priority inheritance*. This technique mandates that a lower-priority task inherit the priority of any higher-priority task pending on a resource they share. This priority change should take place as soon as the high-priority task begins to pend; it should end when the resource is released. This requires help from the operating system.

The second solution, priority ceilings, associates a priority with each resource; the scheduler then transfers that priority to any task that accesses the resource. The priority assigned to the resource is the priority of its highest-priority user, plus one. Once a task finishes with the resource, its priority returns to normal.

A beneficial feature of the priority ceiling solution is that tasks can share resources simply by changing their priorities, thus eliminating the need for semaphores:

```
void TaskA(void)
{
        ...
```

```
        SetTaskPriority(RES_X_PRIO);
        // Access shared resource X.
        SetTaskPriority(TASK_A_PRIO);
        ...
}
```

While Task A's priority is elevated (and it is accessing shared resource X), it should not pend on any other resource. The higher-priority user will only become the highest-priority ready task when the lower-priority task is finished with their shared resource.

While not all of us are writing software for missions to Mars, we should learn from past mistakes and implement solutions that don't repeat them. Many commercial RTOSes include support for either priority inheritance or priority ceilings. Just make sure you enable one.

David Kalinsky is director of customer education at OSE Systems. Earlier in his career, he was involved in the design of medical and aerospace systems. David holds a PhD in nuclear physics from Yale and can be reached by e-mail at david@enea.com.

Michael Barr is the editor in chief of *Embedded Systems Programming.* He holds BS and MS degrees in electrical engineering from the University of Maryland and is a part-time lecturer there. Michael is the author of the best-selling book *Programming Embedded Systems in C and C++.* Contact him at mbarr@cmp.com.

References

Reeves, Glenn. "Re: What Really Happened on Mars?," *Risks-Forum Digest*, Volume 19: Issue 58, January 1998.

Sha L., R. Rajkumar, and J.P. Lehoczky. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization,," *IEEE Transactions on Computers,* September 1990, p. 1175.

Return to the April Table of Contents