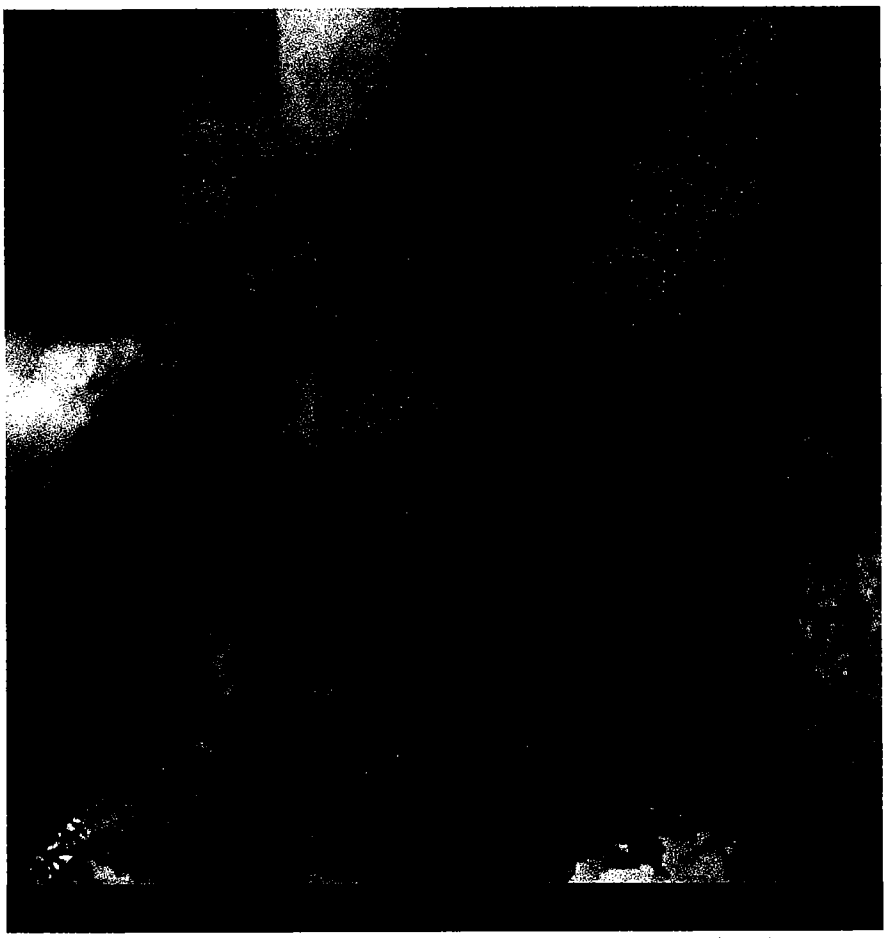
BY NAOMI AVIGDOR

Handling Inverted Priorities



Nance Paternoster

One of the classic problems in real-time system design is the case of priority inversion, where a low priority task can actually block the execution of a high priority task. How you handle this situation will depend on your system design and operating system. One thing is certain, however—you *will* have to address the problem if you work on real-time systems. I'll be using the pSOS+ real-time operating system for the purposes of discussion, but the solution I'll propose is not limited to any particular operating system.

To get our bearings on the problem, let's assume that our system has only three tasks: HighTask, MedTask, and LowTask. These tasks have priorities high, medium, and low, respectively. HighTask and LowTask share a semaphore. MedTask does not need access to that semaphore.

There are two cases to consider. In the first case, LowTask is running and gets the semaphore. While LowTask is holding the semaphore, HighTask becomes ready and needs the same semaphore. At this point, HighTask, which has a higher priority than LowTask, is blocked, waiting for LowTask to finish using the semaphore. In this case, a low priority task is blocking a higher priority task. This case is acceptable because of the shared resource (the semaphore).

At this point, let us consider our second case. LowTask is once again running and gets the semaphore. While LowTask is holding the semaphore, HighTask becomes ready and needs the same semaphore. Now, HighTask is blocked, waiting for LowTask to finish using the semaphore. In this scenario, however, MedTask becomes ready and pre-empts LowTask. MedTask, which does not need the shared semaphore, is

Let LowTask run at the priority of HighTask while it is holding a resource shared with HighTask, just in case HighTask needs it.

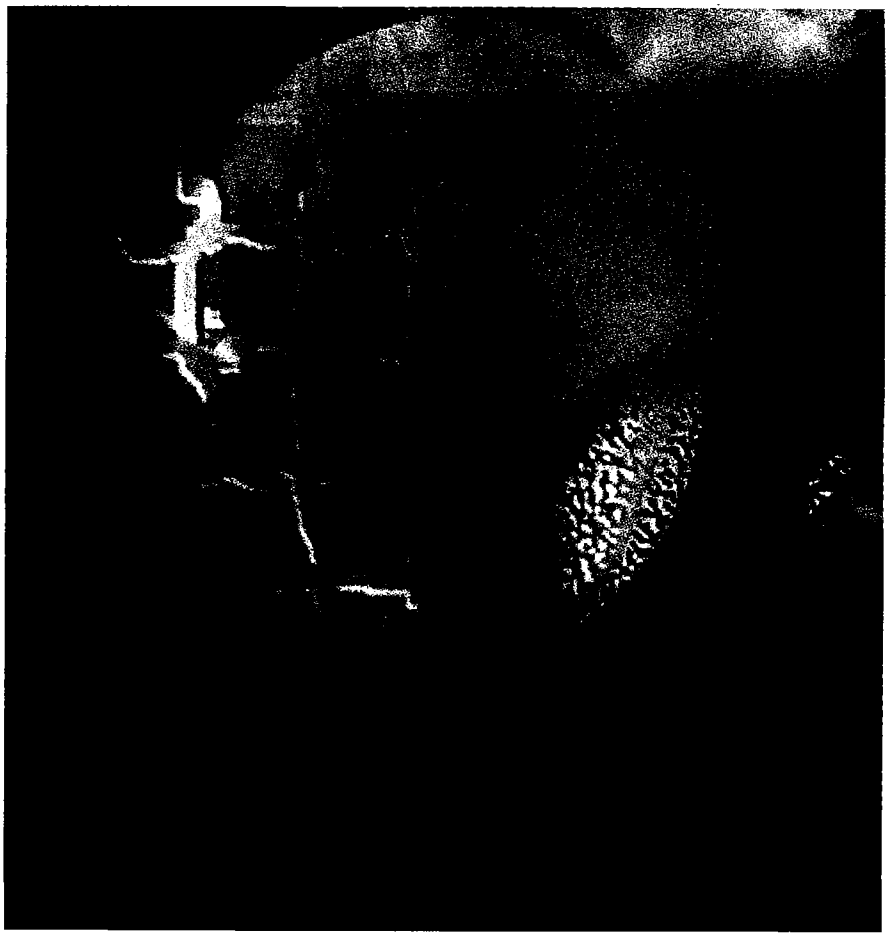
running and blocking Low Task, which is in turn blocking HighTask (by holding the shared semaphore). This condition is known as the classical priority inversion problem.

A correct solution to this problem is one that would not allow MedTask to preempt LowTask, if LowTask is holding a semaphore that might be needed by a higher priority task than MedTask. A correct solution would be to let LowTask run at the priority of HighTask while it is holding a resource shared with HighTask, just in case HighTask may need it while LowTask is holding it.

FINDING A SOLUTION

Since pSOS+ does not assign priorities to semaphores, I propose the following solution to the classical priority inversion problem:

- Create a queue with a single message. This message represents the availability of the semaphore. When the queue is empty then the semaphore is not available, and vice versa.
- Create that queue with the `Q_PRIOR` flag. The highest priority task waiting for the queue message (semaphore) will get it when it becomes available.
- Use one of the 16-byte message areas of the queue message as the keeper of the highest priority level of any task needing that semaphore.
- When each task needing the semaphore is started, or at any other convenient time, the task requests the semaphore (`q_receive`) and examines the highest priority level inside the semaphore message. If it is lower than the task's own priority, update the value to the new higher priority. If the task priority is lower than the one in the message, leave it alone. Put the message back into the queue (`q_send`).
- When a task needs the semaphore, it calls `t_mode` to disable preemption. If the task is the ready task, pSOS+ will continue to run it, even if there are higher priority tasks ready as well.
- The task calls `q_receive` on the semaphore message. Then, when the task gets the semaphore, it calls `t_setpri` with the `TaskId=self=0, NewPriority=ValueFromSemaphoreMessage`. Now the task runs at the semaphore priority level but is still not preemptable.
- The task calls `t_mode` to enable preemption to allow the task to be preemptable by higher priority tasks. This is acceptable because no higher priority



Handling Inverted Properties

task needs the semaphore. At this point, the task that holds the semaphore cannot be blocked by a task with a lower priority than the highest priority semaphore customers, as described in the second case.

■ When the task completes the semaphore use, it calls `t_mode` to disable preemption again, and returns the `semaphore(q_send)`. The task then updates the task priority to normal `operation(t_setpri)` and `t_mode` again to enable preemption.

The call to `t_mode` in the final step may or may not be necessary, depending on the scheduling algorithm used in the particular system. The call may be needed to guarantee the lowering of the task priority immediately following the release of the semaphore.

Another solution would be to proceed as in the first approach, but instead of putting the highest priority level in a queue message, put it in a well-known global place. An example would be an interrupt vector space on a 68xxx family processor. With this solution, there is no need to replace `sm_v` and `sm_p` with `q_calls`. This solution will execute somewhat faster. Both approaches completely eliminate the priority inversion problem.

EVALUATING THE SOLUTIONS

The first question to come to many engineers' minds is the issue of execution time: how much of a penalty are we paying for these approaches? These solutions do add some microseconds to the execution time for each semaphore request and release; `q_receive` takes longer than `sm_p`, `q_send` takes longer than `sm_v` and the two calls for `t_setpri`, and two or four calls for `t_mode`.

The task that holds the semaphore cannot be blocked by a task with a lower priority than the highest priority semaphore customers.

Another issue to consider is the dynamic elevation of semaphore holder priority. Assume that the priorities of all the tasks needing the semaphore are 10, 20, 30, or 40. Every task that gets the semaphore calls `t_setprior` to 40. If at some later time the highest priority task (with priority = 40) does not need the semaphore ever again, the ceiling on the semaphore priority is not dynamically lowered. It is, however, dynamically ele-

Handling Inverted Properties

vated. (See the fourth step outlined previously.)

If the highest priority task ever needing the semaphore is 40, but at a certain point in time, nobody is waiting for the semaphore, why should the semaphore holder's priority be elevated? It should only be elevated when a higher priority task actually needs the semaphore, and only to the level of the "current highest" priority waiting task, not to the "ever highest" priority waiting task. This situation can, of course, be resolved only inside the `sm_p` call inside the kernel because only the kernel has a global view of the system.

WHAT ABOUT THE KERNEL?

Some developers would hold that resolving such difficulties should be the operating system's job. Others would place that responsibility in the hands of the application developer. The question can be stated simply: should the priority inversion problem be resolved at the task level or at the kernel level?

This issue can be resolved at the kernel level, and there are commercially available kernels that do just that. If you believe the kernel is in the proper place, the system designer need not consider the priority inversion problem at all. Sounds like a free lunch, doesn't it? Only the cost of the "free lunch" is overhead (CPU time) at every task switch across the entire system. Even if you have a system with no priority inversion problems, the system bears the overhead anyway because the solution to the problem is at the kernel level.

And what is the cost if a kernel (such as pSOS+) does not take care of the priority inversion problem? The cost is only to the parties involved in the problem, not to all tasks in that system, or for that matter, not to all tasks in any system that has a pSOS+ kernel. So where should the priority inversion problem be solved? My experience leads me to believe that most real-time systems are not affected by the priority inversion problem. The few systems that do have the problem (which, as stated, is called "classical" because it is a valid, well-known problem), should solve the problem outside of the kernel, to minimize the penalty to the majority of other real-time systems using the same kernel.

email: avigdor@cs.umbc.edu