Priority Inheritance: The Real Story

by Doug Locke (July 16, 2002)

In this guest editorial, TimeSys VP of Technology Doug Locke offers a rebuttal to Victor Yodaiken's recently published whitepaper on what's wrong with using a technique called 'priority inheritance' to avoid a problem in real-time systems known as 'priority inversion'.

Introduction

A recent whitepaper by Victor Yodaiken presents a sequence of highly technical arguments regarding the implementability and use of *priority inheritance*, followed by a set of conclusions. The technical arguments in the paper are not new, are generally correct, and have been widely discussed in the real-time research community for many years. However, the conclusions drawn in the paper are badly flawed.

Yodaiken's conclusions are drawn using the classic 'strawman' approach; the paper constructs an artificial hypothesis, and then shows that it might be dangerous. In this case his hypothesis is that priority inheritance is blindly used by engineers as a panacea for all priority inversion problems, and, because it has drawbacks under certain conditions, it is therefore dangerous and should never be used. In other words, because some carpenters might unsafely use a screwdriver to drive nails, screwdrivers should never be used.

Far from being 'incompatible with reliable real-time system design,' priority inheritance is one of a set of important and widely used tools that should be included in any well-designed toolbox for building reliable and responsive time-constrained systems. Like all useful tools, priority inheritance can be used improperly and even dangerously, but when properly implemented and used, it can also make a major contribution to the successful and effective development of real-time systems. The fact that it prevented a potential disaster on the Mars Pathfinder rover (see Footnote 1) is just one illustration of its real-world usefulness as a priority inversion management technique.

Yodaiken's paper also correctly advocates static analysis for systems which have very critical time constraints. Static analysis is not appropriate for every application, but its utilization should be thoroughly understood by anyone involved in hard-real-time or safety-critical applications.

Uncontrolled Priority Inversion

Yodaiken's paper correctly describes priority inheritance as a mechanism used to avoid *priority inversion*. A simple example of priority inversion is seen when a high priority thread needs exclusive access to a resource that is being concurrently accessed by a low priority thread. If one or more medium priority threads then run while the low priority thread has the resource locked, the high priority thread can be delayed indefinitely. Such priority inversion frequently occurs in practical systems. Limiting the adverse effects of priority inversion is extremely important in a system where any kind of predictable response is required.

Avoiding Uncontrolled Priority Inversion

While there are many known academic solutions to the problem of avoiding

BREAKING NEWS
• LinuxDevices launches 2008 Embedded Linux survey
• PXA270 Linux implementation updated
• New PMC vendor debuts Linux-ready cards for military
• Touch-panel PC automates with Linux, Zigbee
• Embedded DRM

Most popular stories -- past 90 days:
• Nokia unveils Linux-powered N810 Internet Tablet
• Low-cost board runs Linux, Google Apps
• Open source, Lego-like computer modules run Linux
• DIY CPU demo'd running Minix
• A developer's perspective on Google's Android SDK
• Top-10 gift ideas for the Linux Gadget Geek
• Asus unveils ultra-low-cost

uncontrolled priority inversion, in industrial practice two techniques are commonly available to user-space applications -- *priority inheritance protocol* and *priority ceiling protocol emulation* (see Footnote 2).

The priority ceiling emulation technique raises the priority of any locking thread to the highest priority of any thread that ever uses that lock (i.e., its priority ceiling). This requires the developer to supply the priority ceiling for each lock. In contrast, priority inheritance will raise the priority of a thread only when holding a lock causes it to block a higher priority thread. When this happens, the low priority thread inherits the priority of the higher priority thread it is blocking.

Applications running in kernel space also have at their disposal the ability to disable interrupts or disable preemption. Disabling interrupts or preemption is an effective, and sometimes necessary, locking technique that avoids unbounded priority inversion, but is of limited general applicability since it stops all other processing. It's like turning all traffic lights red in a city whenever any car wishes to cross any intersection. However, because of its low-overhead, it can work well for locks that are never held for more than a few instructions.

Priority Inheritance vs. Priority Ceiling Emulation

Both priority inheritance protocol and priority ceiling emulation protocol have strengths and weaknesses. When used correctly, they are both useful tools in a real-time designer's toolbox.

Priority ceiling emulation has certain desirable properties -- it has good worst-case priority inversion control, it handles nested locks well, and can avoid deadlock in some cases. Priority inheritance can occasionally lead to poorer worst-case behavior when there are nested locks, and does not avoid deadlocks. However, most performance-critical applications and RTOSs minimize their use of nested locks, and there are other mechanisms to avoid deadlock when nesting cannot be avoided, thereby making priority inheritance an attractive option.

On the other hand, priority inheritance can be implemented such that there is no penalty when the locks are not contended, which covers the vast majority of time-constrained systems. This, in addition to the fact that many extra context switches are avoided, and medium priority tasks are not preempted unnecessarily, leads to excellent average performance. In practical systems, including both hard and soft real-time systems, average performance is as important as worst-case response. In contrast, priority ceiling emulation will pay the cost of changing a thread's priority twice regardless of whether there is contention for the lock or not, resulting in higher overhead and many unnecessary context switches and blocking in unrelated tasks.

Priority ceiling emulation is an attractive choice when the set of threads contending for the lock is known, when there may be nested locks, and when worst-case behavior is the only concern. On the other hand, priority inheritance is very effective when a lock is seldom part of a nested set, and when average performance is relevant in addition to worst-case performance.

Another important aspect to understand is that optimal use of priority ceiling emulation requires static analysis of the system to find the priority ceiling of each lock. While static analysis is highly desirable (even necessary) for many applications with critical response requirements, it may be neither desirable nor cost-effective in many other applications in which only a few parts of the system may be critical. Also, formal real-time analysis is not applicable to systems that are not constructed according to a set of demanding rules. In such cases, priority inheritance is a more effective mechanism since it does not require static analysis.

The Importance of Real-Time Design and Analysis

Priority inheritance and priority ceiling emulation are both effective and powerful techniques to prevent uncontrolled priority inversion when locks are used to protect critical sections in a real-time system. Real-time software designers must make intelligent decisions to use the appropriate technique, depending on their system requirements.

technology supports Linux
• Linux distro revamps for UMPC market
• SPECIAL REPORT: Five years of Motorola Linux phones
• GUI dev tool improves prototyping
• Dual-core VMEbus SBC supports Linux
• Linux mobile group announces SDK strategy
• Mobile Linux stack gains MS file viewer
• Webkit-based browser heads for mobile Linux platform
• Industrial PC bristles with serial, LAN ports
• Mobile multimedia API gains traction
• NVidia enters mobile SoC market -- but where's the Linux?
More . . .

Linux laptop
• Linux-based NAS storage devices expand capacity
• Mini Linux PC breaks $100 barrier
• Penny-sized flash drive holds 16GB

Linux-Watch headlines:
• What's behind the SCO buyout
• Is Microsoft/Yahoo about Windows' failure as a top server platform?
• SCO CEO Darl McBride is on his way out
• SCO goes private, gets $100M to keep going, McBride out?
• Linpus offers a Linux for newbies and experts alike
• The experts' legal guide on free software
• Red Hat: JBoss to capture half of middleware market
• Linux hole patched
• 10 years of open source and counting
• Mozilla dismisses new Firefox flaw warning
More . . .

The solution to priority inversion problems starts with a sound architecture and design that must consider the decomposition of the system into tasks and shared resources, and how they impact the system's ability to meet its timing constraints. Many performance problems can be solved by developing an architecture that avoids unnecessary coupling between tasks through shared resources.

For example, analysis tools (such as TimeWiz) allow real-time system designers to quickly develop a system architecture and understand the impact of their design decisions on timing properties, regardless of the choice of techniques to manage priority inversion. In addition, engineers can take advantage of specialized classes such as those provided by TimeSys to ensure their real-time software architecture and design will meet all time constraints.

Operating systems for performance-critical applications should provide a complete set of tools to manage priority inversion. For example, TimeSys Linux Linux / Real-Time provides priority inheritance mutexes for application developers, and the Fall 2002 release of TimeSys Linux/Real-Time will also offer priority ceiling emulation mutexes.

Can Priority Inheritance Be Successfully Implemented?

Yodaiken's paper correctly notes that implementing priority inheritance can be difficult in complex environments, and implementers not intimately familiar with its nuances can easily make mistakes, but there is ample evidence that it can also be very effectively and correctly implemented and used. For example, in a recent jitter test using TimeSys Linux / Real-Time running on a heavily loaded 1.4GHz Pentium processor (see Footnote 3), the worst-case jitter (i.e., the maximum deviation in the measured period of a periodic task) dropped from 76,492 microseconds to 52 microseconds, simply by enabling priority inheritance in the kernel.

Conclusions

Priority inheritance is one of two basic techniques at the application level for managing priority inversion. This paper has described some of the important criteria for its use by time-constrained applications, but it must be used with care, and it must be implemented correctly. Whatever design choices are made, the architecture and design of the system will be critical to achieving the system's performance goals. Performing a response time analysis (using a tool such as TimeWiz) can significantly aid in ensuring success in meeting performance requirements.

Of course, as Yodaiken's paper notes, the best locking is no locking. There are four simple rules for protecting resources:

1. Using non-locking atomic operations (such as the flip-buffer) to avoid locking is clearly the best approach.
2. When a lock must be acquired, it should be held for as short a time as possible.
3. When locks are used, training classes such as those provided by TimeSys, for example, will ensure that engineers understand the limitations of each of the various methods for managing priority inversion.
4. When selecting an operating system, ensure that it provides all the tools for managing priority inversion, so you have the flexibility to meet a wide range of performance requirements.

Any full-featured RTOS should include a full range of Design and Development tools, Operating System, Software Development Kits, and Training to design and deploy embedded systems supporting the full range of performance requirements. In addition, they should provide a selection of tools for managing priority inversion, permitting the application engineers to choose the best approach for their applications, and enabling the construction of reliable, robust, and predictable time-constrained systems. This includes both priority inheritance mutexes and priority ceiling emulation mutexes (the latter will be available from TimeSys in Fall, 2002). Coupled with visualization, analysis, and simulation tools such as TimeWiz and TimeTrace to see all the scheduling, locking, and other key system events, everything needed to meet a wide range of performance

requirements, from hard real-time to soft real-time, and even non-real-time, should be available for the design engineer.

Footnotes:

1. Details on the Mars Pathfinder priority inversion problem are available here.
2. Often incorrectly referred to as *priority ceiling protocol*. The original priority ceiling protocol, described in a paper by Sha, Lehoczky, and Rajkumar, is expensive to implement and not available in any commercial operating system. PCP emulation (also known as the highest locker protocol) is a simplified version of the original PCP protocol and is part of various standards such as POSIX (POSIX_PRIO_PROTECT), the Real-Time Specification for Java (RTSJ), OSEK, and the Ada 95 real-time specifications. Priority inheritance is also part of standards such as POSIX (POSIX_PRIO_INHERIT), and the RTSJ.
3. 1.4 GHz is somewhat faster that is usually available in embedded systems, but even with this speed, the worst-case jitter is still over 70 milliseconds! This illustrates that throwing fast hardware at the application doesnt solve the predictability problem, but avoiding priority inversion does.

About the author: Doug Locke, Vice President of Technology of TimeSys Corporation, has spent more than 35 years intimately involved in the specification, architecture, design, and implementation of time-critical systems spanning a wide range of applications including industrial control, space (both ground based and flight), avionics, command & control, and automotive. His technical interests cover real-time operating systems as well as real-time systems architecture, design, implementation, analysis, standards, operating systems, and languages. Doug has served, and continues to serve, on various standardization committees related to real-time, including POSIX, Real-Time CORBA, Real-Time UML and the Real-Time Specification for Java. Prior to joining TimeSys, he was Chief Scientist for Lockheed Martin's Information Solutions organization, with technical oversight responsibility for many performance-critical and safety-critical projects. He holds a PhD in Computer Science from Carnegie Mellon University, with a dissertation on real-time scheduling.

Talk back! Do you have questions or comments on this article? talkback here

(Click here for further information)