SMP and Embedded Realtime Myths

Paul E. McKenney
Linux Technology Center
IBM Beaverton
`paulmck@us.ibm.com`

October 9, 2006

Abstract

With the advent of multi-threaded/multi-core CPUs, even embedded realtime applications are starting to run on SMP systems — for example, both the Xbox 360 and PS3 are multi-threaded, and there have even been SMP *ARM* processors! As this trend continues, there will be an increasing need for realtime response from SMP systems. Because not all embedded systems vendors will be willing or able to create or purchase SMP realtime operating systems, we can expect that a number of them will make use of Linux®.

Because of this change, a number of realtime tenets have now become myths. This article exposes these myths, and then discusses some of the challenges that Linux is surmounting in order to meet the needs of this new SMP-realtime-embedded world.

Realtime Myths

New technologies often have a corrosive effect on the wisdom of the ages. The advent of commodity multi-core and multi-threaded hardware is no different, making myths of the following pearls of wisdom:

1. Embedded systems are always uniprocessor systems.
2. Parallel programming is mind-crushingly difficult.
3. Realtime must be either hard or soft.
4. Parallel realtime programming is impossibly difficult.
5. There is no connection between realtime and enterprise systems.

Each of these myths is exposed in the following sections, and Ingo Molnar's -rt realtime patchset (also known as the "`CONFIG_PREEMPT_RT` patchset" after the configuration variable used to enable realtime behavior) plays a key role in exposing the last two myths.

Myth 1: Embedded Systems are Always Uniprocessor Systems

Past embedded systems were almost always uniprocessors, especially given that single-chip multiprocessors are a very recent phenomonon. The PS3, the Xbox 360, and the SMP ARM are recent exceptions to this rule. But what does the future hold?

Figure 1 shows how clock frequencies have leveled off since 2003. Now, Moore's Law is still in full force, as transistor densities are still increasing. However, these increasing densities are no longer providing the side-benefit of increased clock frequency that they once did.

Some say that parallel processing, hardware multithreading, and multi-core CPU chips will be needed to make good use of the ever-increasing numbers of transistors. Others say that embedded systems need increasing levels of integration and reduced power

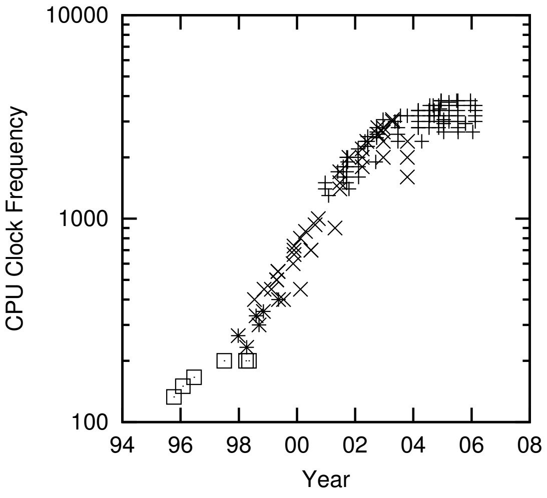

Figure 1: Clock-Frequency Trend for Intel®CPUs

consumption more than they do ever-increasing performance. Embedded systems vendors might therefore choose more on-chip I/O or memory over increased parallelism.

This debate will not be resolved soon, though we have all seen examples of multi-threaded and multi-core CPUs in embedded systems. That said, as multi-threaded/multi-core systems become cheaper and more prevalent, we will see more rather than fewer of them.

But these multi-threaded/multi-core systems require parallel software. Given the forbidding reputation of parallel programming, how are we going to program these systems successfully?

Myth 2: Parallel Programming is Mind-Crushingly Difficult

Why is parallel programming hard?

Answers include deadlocks, race conditions, and testing coverage, but the real answer is that *it is not really all that hard.* After all, if parallel programming was really so difficult, why are there so many parallel open-source projects, including Apache, MySQL, and the Linux kernel?

A better question would be: "Why is parallel programming *perceived* to be so difficult?" Let's go back to the year 1991. I was walking across the parking lot to Sequent's benchmarking center carrying six dual-80486 CPU boards, when I suddenly realized that I was carrying several times the price of my house. (Yes, I *did* walk more carefully. Why do you ask?) These horribly expensive systems were limited to a privileged few, who were the only ones with the opportunity to learn parallel programming

In contrast, in 2006, I am typing on a dual-core x86 laptop that is orders of magnitude cheaper than even one of Sequent's CPU boards. Because almost everyone can now gain access to parallel hardware, almost everyone can learn to program it, and also learn that although it can be non-trivial, it is really not all that hard.

Even so, many multi-threaded/multi-core embedded systems have realtime constraints. But what exactly *is* realtime?

Myth 3: Realtime Must Be Either Hard or Soft

There is hard realtime, which offers unconditional guarantees, and there is soft realtime, which does not. What else do you need to know?

As it turns out, quite a bit. There are at least four different definitions of "hard realtime". Needless to say, it is important to understand which one your users have in mind.

In one definition of hard realtime, the system must *always* meet its deadlines. However, if you show me a hard realtime system, I will show you the hammer that will cause it to miss its deadlines, as shown in Figure 2.

Of course, this is unfair. After all, we cannot blame software for hardware failures that it did not cause. Therefore, in another definition of hard realtime, the system must always meet its deadlines, but only in absence of hardware failure. This divide-and-conquer

Figure 2: Hard Realtime: But I Have a Hammer

approach can simplify things, but, as shown in Figure 3, is not sufficient at the system level. Nonetheless, this definition can be useful given restrictions on the environment, including:

1. Interrupt rates
2. Cache misses
3. Memory-system overhead due to DMA
4. Memory-error rate in ECC-protected systems
5. Packet-loss rate in systems requiring networking

If these restrictions are violated, the system is permitted to miss its deadlines. For example, if a hyperactive interrupt system delivered an interrupt after each instruction, the appropriate action might be to replace the broken hardware rather than coding around it. After all, if this degenerate situation must be accounted for, the latencies will likely become uselessly long. Alternatively, "diamond hard" realtime operating systems and applications might run with interrupts disabled, giving up compatibility with off-the-shelf software in order to gain additional robustness in face of hardware failure.

In yet another definition of hard realtime, the system is allowed to miss its deadline, but only if it announces its failure within the deadline specified. This

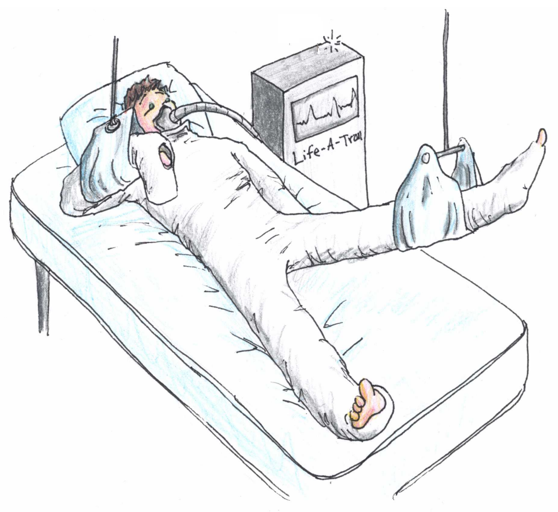

Figure 3: Hard Realtime: Sometimes System Failure is not an Option!

sort of definition can be useful in data-fusion applications. For example, a system might have a high-precision location sensor with unpredictable processing overhead as well as a rough-and-ready location sensor with deterministic processing overhead. A reasonable hard-realtime strategy would be to give the high-precision sensor a fixed amount of time to get its job done, and if it fails to do so, abort its calculation, relying instead on the rough-and-ready sensor. However, one (useless) way to meet the letter of this definition would be to unconditionally announce failure, as illustrated by Figure 4. Clearly, a useful system would almost always complete its work in time (and this observation applies to soft-realtime systems as well).

Finally, some define hard realtime with a test suite: a system passing the test is labelled hard realtime. Purists might object, demanding instead a mathematical proof. However, given that proofs can be subject to error, especially for today's complex systems, a test suite can be an excellent additional proof point. I certainly do not wish to put *my* life at the mercy of untested software!

This is not to say that hard realtime is undefined

Did you complete your project?
No.
Have you started it?
Nope.
...Is the planning done?
Huh-uh.
What use are you!?
Hey, I never have missed a deadline without letting people know!

Figure 4: Hard Realtime: At Least I Let You Know!

or useless. Instead, "hard realtime" is the start of a conversation rather than a complete requirement. You should ask the following questions:

1. Which operations must provide hard-realtime response? (For example, I have yet to run across a requirement for realtime filesystem unmounting.)
2. What is the deadline? A ten-millisecond deadline is one thing, a one-microsecond deadline quite another.
3. What is to happen in case of hardware failure?
4. What is the required probability of meeting that deadline? (For hard realtime, this will be 100%.)
5. What degradation of non-realtime performance, throughput, and scalability can be tolerated?

One piece of good news is that realtime deadlines that once required extreme measures are now easily met with off-the-shelf hardware and open-source software, courtesy of Moore's Law.

But what if your realtime application is to run on an embedded multi-core/multi-threaded system? How can you deal with both realtime deadlines *and* parallel programming?

Myth 4: Parallel Realtime Programming is Impossibly Difficult

Parallel programming might not be mind-crushingly hard, but it is certainly harder than single-threaded programming. Realtime programming is also hard. So why would anyone be crazy enough to take on both at the same time?

It is true that realtime parallel programming poses special challenges, including interactions with lock-induced delays, interrupt handlers, and priority inversion. However, Ingo Molnar's -rt patchset provides both kernel and application developers with tools to deal with these challenges. These tools are described in the following sections.

Locking and Realtime Latency

Much ink has been spilled on locking and realtime latency, but we will stick to the following simple points:

1. Reducing lock contention improves SMP scalability *and* reduces realtime latency.
2. When lock contention is low, there are a finite number of tasks, critical-section execution time

is bounded, and locks act in a first-come-first-served manner to the highest-priority tasks, then lock wait times for those tasks will be bounded.

3. An SMP Linux kernel by its very nature requires very few modifications in order to support the aggressive preemption required by realtime.

The first point should be obvious, since spinning on locks is bad for both scalability *and* latency. For the second point, consider a queue at a bank where each person spends a bounded time T with a solitary teller, there are a bounded number of other people N, and the queue is first-come-first-served. Because there can be at most N people ahead of you, and each can take at most time T, you will wait for at most time NT. Therefore, FIFO priority-based locking really can provide hard realtime latencies.

For the third point, see Figure 5. The left-hand side of the diagram shows three functions A(), B(), and C() executing on a pair of CPUs. If functions A() and B() must exclude function C(), then some sort of locking scheme must be used. However, that same locking provides the protection needed by the -rt patchset's preemption, as shown on the right-hand side of this diagram. If function B() is preempted, then function C() blocks as soon as it tries to acquire the lock, which permits B() to run. After B() completes, C() may acquire the lock and resume running.

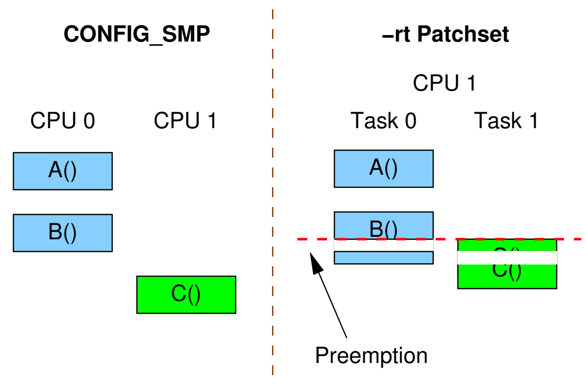

Figure 5: SMP Locking and Preemption

This approach requires that kernel spinlocks block, and this change is fundamental to the -rt patchset. In addition, per-CPU variables must be protected more rigorously. Interestingly enough, the -rt patchset also located a number of SMP bugs that had gone undetected.

However, in the standard Linux kernel, interrupt handlers cannot block. But interrupt handlers must acquire locks, which can block in -rt. What can be done?

Interrupt Handlers

Not only are blocking locks a problem for interrupt handlers, they can seriously degrade realtime latency, as shown in Figure 6.

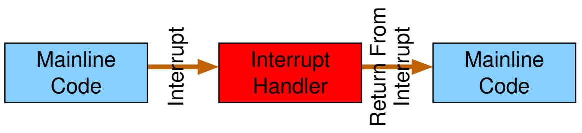

Figure 6: Interrupts Degrade Latency

This degradation can be avoided by running the interrupt handler in process context, as shown in Figure 7, which also allows them to acquire blocking locks. Even better, these process-based interrupt handlers can actually be preempted by user-level realtime threads, as shown in Figure 8, where the blue rectangle within the interrupt handler represents a high-priority realtime user process preempting the interrupt handler.

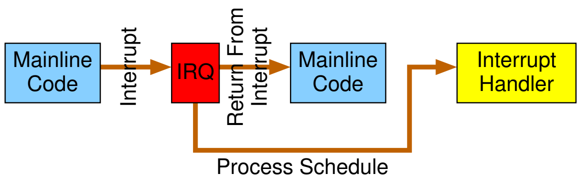

Figure 7: Move Interrupt Handlers to Process Context

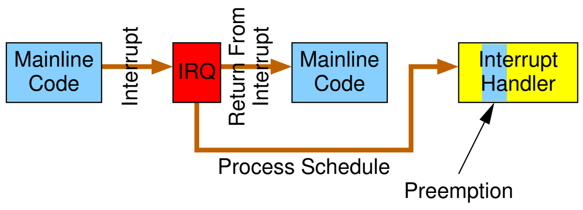

Figure 8: Preempting Interrupt Handlers

Of course, "with great power comes great responsibility". For example, a high-priority realtime user process could starve interrupts entirely, shutting down all I/O. One way to handle this situation is to provide a low-priority "canary" process. If the "canary" is blocked for longer than a predetermined time, one might kill the offending thread.

Running interrupts in process context permits interrupt handlers to acquire blocking locks, which in turn allows critical sections to be preempted, which permits extremely fast realtime scheduling latencies. In addition, the -rt patchset permits realtime application developers to select the realtime priority at which interrupt handlers run. By running only the most critical portions of the realtime application at higher priority than the interrupt handlers, the developers can minimize the amount of code for which "great responsibility" must be shouldered.

However, preempting critical sections can lead to priority inversion, as described in the next section.

Priority Inversion

Priority inversion is illustrated by Figure 9. A low-priority process P2 holds a lock, but is preempted by medium-priority processes. When high-priority process P1 tries to acquire the lock, it must wait, because P2 holds it. But P2 cannot release it until it runs again, which will not happen while the medium-priority processes are running. So, in effect, the medium-priority processes have blocked a high-priority process: in short, priority inversion.

One way to prevent priority inversion is to disable preemption during critical sections, as is done in `CONFIG_PREEMPT` builds of the Linux kernel. However, this preemption disabling can result in excessive latencies.

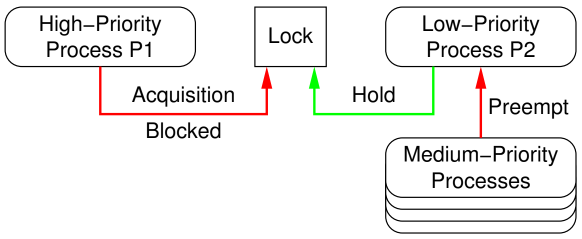

Figure 9: Priority Inversion

The -rt patchset therefore uses priority inheritance instead, so that P1 donates its priority to P2, but only for as long as P2 continues to hold the lock, as shown in Figure 10. Because P2 is now running at high priority, it preempts the medium-priority processes, completing its critical section quickly and then handing the lock off to P1.

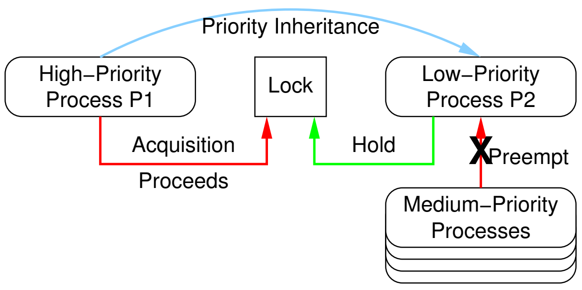

Figure 10: Priority Inheritance

So priority inheritance works well for exclusive locks, where only one thread can hold the lock at a given time. But there are also reader-writer locks, which can be held by one writer on the one hand or by an unlimited number of readers on the other. The fact that a reader-writer lock can be held by an unlimited number of readers can be a real problem for priority inheritance, as illustrated in Figure 11. Here, several low-priority processes are read-holding lock L1, but are preempted by medium-priority processes. Each low-priority process might also be blocked write-acquiring other locks, which might be read-held by even more low-priority processes, all of which are also preempted by the medium-priority processes.

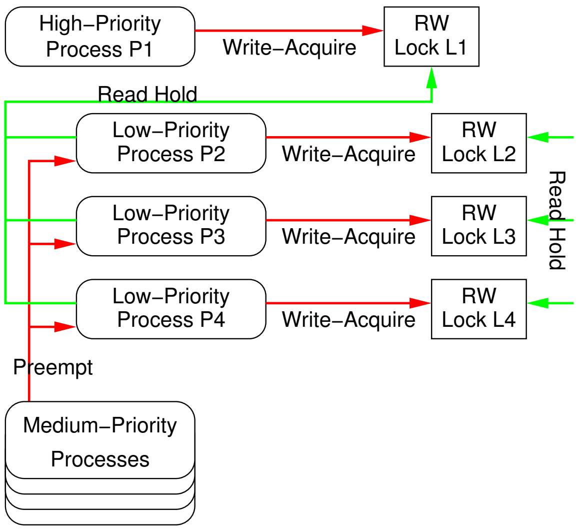

Figure 11: Reader-Writer Lock Priority Inversion

Priority inheritance can solve this problem, but the cure is worse than the disease. For example, the arbitrarily bushy tree of preempted processes requires complex and slow bookkeeping. But even worse, before the high-priority writer can proceed, *all* of the low-priority processes must complete their critical sections, which will result in arbitrarily long delays.

Such delays are *not* what we want in a realtime system. This situation resulted in numerous "spirited" discussions on the Linux-kernel mailing list, which Ingo Molnar closed down with the following proposal:

1. Only one task at a time may read-acquire a given reader-writer lock.

2. If #1 results in performance or scalability problems, the problematic lock will be replaced with RCU (read-copy update).

RCU can be thought of as a reader-writer lock where readers never block; in fact, readers execute a deterministic number of instructions. Writers have much higher overhead, as they must retain old versions of the data structure that readers might still be referencing. RCU provides special primitives to allow writers to determine when all readers have completed, so that the writer can determine when it is safe to free old versions. RCU works best for read-mostly data structures, or for data structures with hard-realtime readers. (More detail may be found at `http://en.wikipedia.org/wiki/RCU` and even more detail may be found at `http://www.rdrop.com/users/paulmck/RCU/`. Although user-level RCU implementations have been produced for experimental purposes, for example `http://www.cs.toronto.edu/~tomhart/perflab/ipdps06.tgz`, production-quality RCU implementations are currently found only in kernels. Fixing this is on my to-do list.)

A key property of RCU is that writers never block readers, and, conversely, readers do not block writers from modifying a data structure. Therefore, RCU cannot cause priority inversion. This is illustrated in Figure 12. Here, the low-priority processes are in RCU read-side critical sections, and are preempted by medium-priority processes, but because the locks are used only to coordinate updates, the high-priority process P1 can immediately acquire the lock and carry out the update by creating a new version. Freeing the old version *does* have to wait for readers to complete, but this freeing can be deferred to avoid degrading realtime latencies.

This combination of priority inheritance and RCU permits the -rt patchset to provide realtime latencies on mid-range multiprocessors. But priority inheritance is not a panacea. For example, one could imagine applying some form of priority inheritance to real live users who might be blocking high-priority processes, as shown in Figure 13. However, I would rather we did not.

Parallel Realtime Programming Summary

I hope I have convinced you that the -rt patchset greatly advances Linux's parallel realtime capabilities, and that Linux is quickly becoming capable of supporting the parallel relatime applications that are appearing in embedded environments. Parallel real-

MR. McKENNEY, YOU'RE HOLDING UP A REALTIME PROCESS.
SO?
SO I'M BOOSTING YOUR PRIORITY.
©2005

Figure 13: Priority Boosting for Users?

time programming is decidedly non-trivial, in fact, many exciting challenges lie ahead in this field, but it is far from impossible.

But there are a number of realtime operating systems, and some even provide some SMP support. What is special about realtime Linux?

Myth 5: There is no Connection Between Realtime and Enterprise Systems

To test the fifth and final myth, and to show just what is special about realtime Linux, let's first outline the -rt patchset's place in the realtime pantheon.

The -rt patchset turns Linux into an extremely capable realtime system. Is Linux suited to all purposes? The answer is clearly "no", as can be seen from Figure 14. With the -rt patchset, Linux can achieve scheduling latencies down to a few tens of microseconds — an impressive feat, to be sure, but some applications need even more. Systems with very tight hand-coded assembly-language loops might achieve sub-microsecond response times, at which point memory and I/O-system latencies loom large. Below this point comes the realm of special-purpose digital hardware, and below that the realm of analog microwave and photonics devices.

However, Linux's emerging realtime capabilities are sufficient for the vast majority of realtime applications. Furthermore, Linux brings other strengths to the realtime table, including full POSIX semantics, a complete set of both open-source and proprietary applications, a high degree of configurability, and a vibrant and productive community.

In addition, realtime Linux forges a bond between the realtime and enterprise communities. This bond will become tighter as enterprise applications face increasing realtime requirements. These requirements are already upon us: for but one example, web retailers find that they lose customers when response times extend beyond a few seconds. A few seconds

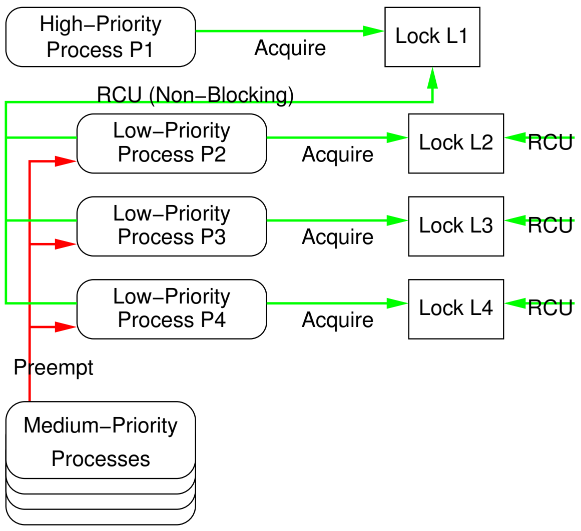

Figure 12: RCU Prevents Priority Inversion

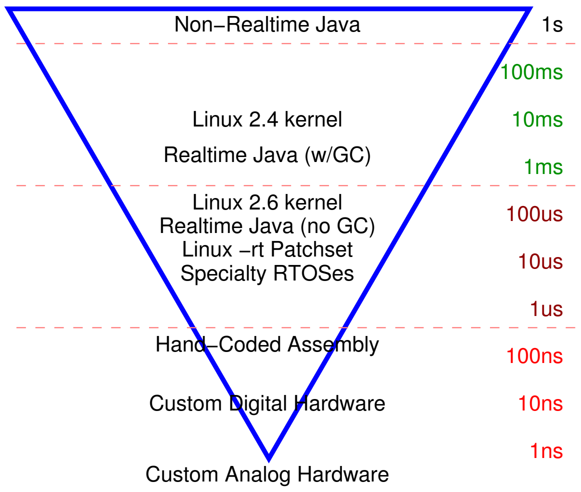

Figure 14: Realtime Capability Triangle

might seem like a long time, but not when you (1) subtract off typical Internet round-trip times and (2) divide by an increasingly large numbers of layers, including firewalls, IP load levellers, web servers, web-application servers, XML accelerators, and database servers — across multiple organizations. The required per-machine response times fall firmly into realtime territory.

Web2.0 mashups will only increase the pressure on per-machine latencies, since such mashups must gather information from multiple web sites, so that the slowest site controls the overall response time. This pressure will be most severe in cases where information gathered from one site is used to query other sites, thus serializing the latencies.

We are witnessing nothing less than the birth of a new kind of realtime: enterprise realtime. What exactly is "enterprise realtime"? Enterprise realtime is defined by developer and user requirements, as might be obtained from the realtime questions listed in the discussion of Myth 3. Some of these requirements would specify latencies and guarantees (hard or soft) for various operations, while others would surround the ecosystem, where realtime Linux's rich array of capabilities, environments, applications, and supported hardware really shines.

Of course, even the rich realtime-Linux ecosystem cannot completely remove the need for special-purpose hardware and software. However, the birth of enterprise realtime will provide a new-found ability to share software between embedded and enterprise systems. Such sharing will greatly enrich both environments.

Future Prospects

Impressive as it is, realtime Linux with the -rt patchset focuses primarily on user-process scheduling and inter-process communication. Perhaps the future holds realtime protocol stacks or filesystems, and perhaps also greater non-realtime performance and scalability while still maintaining realtime response, allowing electrical power to be conserved by consolidating realtime and non-realtime workloads onto a single system.

However, realtime applications and environments are just starting to appear on Linux, both from proprietary vendors and F/OSS communities. For example, existing realtime Java™ environments require that realtime programs avoid the garbage collector, making it impossible to use Java's standard runtime libraries. IBM recently announced a Java JVM that meets realtime deadlines even when the garbage collector is running, allowing realtime code to use standard libraries. This JVM is expected to greatly ease coding of realtime systems, and to ease conversion of older realtime applications using special-purpose languages such as ADA.

In addition, there are realtime audio systems, SIP servers, and object brokers, but much work remains to provide a full set of realtime webservers, web application servers, database kernels, and so on. Realtime applications and environments are still few and far between.

I very much look forward to participating in — and making use of — the increasing SMP-realtime capability supported by everyday computing devices!

Acknowledgements

No article mentioning the -rt patchset would be complete without a note of thanks to Ingo Molnar, Thomas Gleixner, Sven Deitrich, K. R. Foley, Gene Heskett, Bill Huey, Esben Neilsen, Nick Piggin, Steven Rostedt, Michal Schmidt, Daniel Walker, and Karsten Wiese. I also owe a debt of gratitude to Ted Ts'o, Darren Hart, Dinakar Guniguntala, John Stultz, Vernon Mauery, Jennifer Monk, Sripathi Kodi, Tim Chavez, Vivek Pallantla, and Hugh Miller for many valuable realtime-Linux words and deeds. I am likewise grateful to David Bacon and his realtime-GC-research team and to Boas Betzler for many productive conversations. We all owe Bruce Jones, John Kacur, and Mark Brown many thanks for their invaluable service rendering this article human-readable. Finally, many thanks go to Daniel Frye for his unstinting support of this effort.

Legal Statement