SMP and Embedded Real Time

Paul E. McKenney

Abstract

As embedded real-time applications start to run on SMP systems, kernel issues emerge.

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

Because of this change, a number of real-time 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-real-time-embedded world.

Real-Time Myths

New technologies often have a corrosive effect on the wisdom of the ages. The advent of commodity multicore and multithreaded 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. Real time must be either hard or soft.
4. Parallel real-time programming is impossibly difficult.
5. There is no connection between real-time and enterprise systems.

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

Myth 1: Embedded Systems Are Always Uniprocessor Systems

Past embedded systems almost always were uniprocessors, especially given that single-chip multiprocessors are a very recent phenomenon. 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.

![Figure 1. Clock-Frequency Trend for Intel CPUs](image)

Some say that parallel processing, hardware multithreading and multicore 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 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, although we have all seen examples of multithreaded and multicore CPUs in embedded systems. That said, as multithreaded/multicore systems become cheaper and more prevalent, we will see more rather than fewer of them.

But these multithreaded/multicore 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 now can gain access to parallel hardware, almost everyone can learn to program it and also learn that although it can be nontrivial, it is really not all that hard.
Even so, many multithreaded/multicore embedded systems have real-time constraints. But what exactly is real time?

**Myth 3: Real Time Must Be Either Hard or Soft**

There is hard real time, which offers unconditional guarantees, and there is soft real time, 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 real time. Needless to say, it is important to understand which one your users have in mind.

In one definition of hard real time, the system always must meet its deadlines. However, if you show me a hard real-time 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 real time, the system always must meet its deadlines, but only in absence of hardware failure. This divide-and-conquer approach can simplify things, but, as shown in Figure 3, it 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.
Figure 3. Hard real time: sometimes system failure is not an option!

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 code around it. After all, if this degenerate situation must be accounted for, the latencies will likely become uselessly long. Alternatively, “diamond hard” real-time 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 real time, the system is allowed to miss its deadline, but only if it announces its failure within the deadline specified. This 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 real-time 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 announce failure unconditionally, as illustrated by Figure 4. Clearly, a useful system almost always would complete its work in time (and this observation applies to soft real-time systems as well).
Finally, some define hard real time with a test suite: a system passing the test is labeled hard real time. 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 real time is undefined or useless. Instead, “hard real time” is the start of a conversation rather than a complete requirement. You should ask the following questions:

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

One piece of good news is that real-time 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 real-time application is to run on an embedded multicore/multithreaded system? How can you deal with both real-time deadlines and parallel programming?

**Myth 4: Parallel Real-Time Programming Is Impossibly Difficult**

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

It is true that real-time 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 Real-Time Latency**

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

1. Reducing lock contention improves SMP scalability and reduces real-time 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 real time.

The first point should be obvious, because 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 real-time 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 $C()$, 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, 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](image)

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, but they also can seriously degrade real-time latency, as shown in Figure 6.

![Figure 6. Interrupts Degrade Latency](image)

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.

![Figure 7. Move Interrupt Handlers to Process Context](image)

Even better, these process-based interrupt handlers can actually be preempted by user-level real-time threads, as shown in Figure 8, where the blue rectangle within the interrupt handler represents a high-priority real-time user process preempting the interrupt handler.
Of course, “with great power comes great responsibility.” For example, a high-priority real-time 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 real-time scheduling latencies. In addition, the -rt patchset permits real-time application developers to select the real-time priority at which interrupt handlers run. By running only the most critical portions of the real-time 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.

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.

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.

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 real-time 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 real-time readers. More detail may be found at [http://en.wikipedia.org/wiki/RCU](http://en.wikipedia.org/wiki/RCU), and even more detail may be found at [http://www.rdrop.com/users/paulmck/RCU](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](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 immediately can 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 real-time latencies.

![Figure 12. RCU Prevents Priority Inversion](image)

This combination of priority inheritance and RCU permits the -rt patchset to provide real-time 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.

![Figure 13. Priority Boosting for Users?](image)

Parallel Real-Time Programming Summary

I hope I have convinced you that the -rt patchset greatly advances Linux's parallel real-time capabilities, and that Linux is quickly becoming capable of supporting the parallel real-time applications that are appearing in embedded environments. Parallel real-time programming is decidedly nontrivial. In fact, many exciting challenges lie ahead in this field, but it is far from impossible.

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

Myth 5: There Is No Connection between Real-Time and Enterprise Systems

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

The -rt patchset turns Linux into an extremely capable real-time 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 real-time capabilities are sufficient for the vast majority of real-time applications. Furthermore, Linux brings other strengths to the real-time 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, real-time Linux forges a bond between the real-time and enterprise communities. This bond will become tighter as enterprise applications face increasing real-time requirements. These requirements are already upon us – for example, Web retailers find that they lose customers when response times extend beyond a few seconds. A few seconds 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 levelers, Web servers, Web-application servers, XML accelerators and database servers -- across multiple organizations. The required per-machine response times fall firmly into real-time territory.

Web 2.0 mashups will only increase the pressure on per-machine latencies, because 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 when 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 real time: enterprise real time. What exactly is enterprise real time? Enterprise real time is defined by developer and user requirements, as might be obtained from the real-time 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 real-time Linux’s rich array of capabilities, environments, applications and supported hardware really shine.

Of course, even the rich real-time-Linux ecosystem cannot completely remove the need for special-purpose hardware and software. However, the birth of enterprise real time 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, real-time Linux with the -rt patchset focuses primarily on user-process scheduling and interprocess communication. Perhaps the future holds real-time protocol stacks or filesystems, and perhaps also greater non-real-time performance and scalability while still maintaining real-time response, allowing electrical power to be conserved by consolidating real-time and non-real-time workloads onto a single system.

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

In addition, there are real-time audio systems, SIP servers and object brokers, but much work remains to provide a full set of real-time Web servers, Web application servers, database kernels and so on. Real-time applications and environments are still few and far between.

I very much look forward to participating in -- and making use of -- the increasing SMP-real-time 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 Nielsen, Nick Piggot, Steven Rostedt, Michal Schmidt, Daniel Walker and Karsten Wiese. I also owe a debt of gratitude to Ted Tvo, Darren Hart, Dinakar Guniguntala, John Stultz, Vernon Maury, Jennifer Monk, Sripathi Kodi, Tim Chavez, Vivek Pallantla and Hugh Miller for many valuable real-time-Linux words and deeds. I am likewise grateful to David Bacon and his real-time-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**

This work represents the view of the author and does not necessarily represent the view of IBM.

Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc., in the United States, other countries or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries or both.

Other company, product or service names may be trademarks or service marks of others.