
It is really time to celebrate!
25th anniversary of RTLWS
20th anniversary of Preempt-RT

Time advances with a constant pace, but it passes quickly.

In 1999, the first Real-Time Linux Workshop (RTLWS) took place in Vienna. In 2004, the Linux Real-Time debate unfolded on the Linux kernel mailing list, which marks the start of the Preempt-RT project.

These anniversaries are definitely worth coming together to commemorate. They also provide a worthwhile opportunity to take a look back on 25 years of history. This anniversary publication is intended to provide insight into the history of Linux Real-Time and to honor the people who were involved and have contributed to this effort. We hope you will enjoy this journey down the memory lane.

Uhldingen-Mühlhofen, September 2024

Thomas Gleixner and Heinz Egger

Table of contents

Looking back at the evolution of Linux Real-Time	1
Important steps of Preempt-RT	18
Retrospectives	29
Real-Time quotes	37
The RTLWS archives	41
Obituaries	47
About Linutronix	51
Imprint	53

Trademark information

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

UNIX is a registered trademark of The Open Group.

WINDOWS is a registered trademark of Microsoft Corporation.

Other products mentioned may be trademarks of their respective corporations.

Looking back at the evolution of Linux Real-Time

Commemorating the 25th anniversary of the first Real-Time Linux Workshop (RTLWS) and the 20th anniversary of the Linux Real-Time preemption patch (Preempt-RT) provides a worthwhile opportunity to look back on the history of Linux Real-Time.

What is Real-Time?

The ISO 2382 standard defines "Real-Time" as the capability of a system to respond to inputs or events within a specified time frame, known as the deadline. The standard distinguishes between Hard and Soft Real-Time systems. Hard Real-Time systems must guarantee deterministic behavior, where a violation of the deadline is considered a catastrophic failure. Soft Real-Time systems can violate the deadline guarantees occasionally, however the system has to be designed to handle violations gracefully without causing fatal consequences.

Hard Real-Time systems are used in control systems, avionics, automotive and other areas including in the context of functional safety. For functional safety deployments, Hard Real-Time systems have to be designed carefully and under certain circumstances, the correctness has to be verified against a formal model of the system.

General purpose and Real-Time operating systems

A General Purpose Operating System (GPOS) is designed to handle a wide range of applications, primarily optimized for throughput and compute performance. However, it does not provide guarantees for response time to inputs or events. GPOSes are feature rich and not targeted to a specific application space.

Contrary to that, Real-Time Operating Systems (RTOS) are designed for use cases which require deterministic response times. They are used in control systems and other areas, where deterministic behavior is a critical part of the overall system design requirements. A RTOS is typically tailored to the requirements of the application and contains only the minimum set of required features.

Historically, RTOSes were implemented for specific microcontrollers, while broader computing tasks were offloaded to a GPOS on a separate processor system. This has the advantage that formal verification is restricted to the RTOS part, but it severely limits the ability of communication and data exchange between the Real-Time and non Real-Time parts. In certain application areas, e.g. automotive, this led to an aggregation of a large number of individual nodes connected through Real-Time aware communication buses, which significantly increased the complexity of assessing the overall system behavior for correctness.

With the availability of powerful commodity processors for general computing, the industry looked for ways to consolidate such setups so that Real-Time and general compute tasks can be handled by the same processor. While this makes overall system validation more complex, it allows the implementation of more resource demanding Real-Time applications and removes the restrictions on communication and data exchange. Aside from this consolidation potential, the increasing connectivity of Real-Time systems adds security requirements, which are more advanced in GPOSes than in RTOS implementations.

Linux - the early days

On August 25th, 1991, Linus Torvalds announced that he was working on a free operating system for 386 (486) AT clones as a hobby project. The famous postscript of the email said:

> *It is NOT portable (uses 386 task switching etc),*
> *and it probably never will support anything other than*
> *AT-harddisks, as that's all I have :-(.*

Neither Torvalds nor any participant on the comp.os.minix Usenet newsgroup could have anticipated that Linux would grow into one of the most successful and influential software projects, supporting more architectures than any other operating system.

Early Linux Real-Time research

The rapidly growing popularity of Linux in general computing got the embedded industry interested, but the lack of Real-Time capabilities was holding adoption off.

Linux gained traction in the academic space because it was easily accessible, free of cost, and less complex than the BSD variants. This allowed Real-Time researchers to utilize Linux as the base for their research and experiments. This also lifted research restrictions as commercial RTOSes were hard to access and often put strict limitations on the ability to publish data. Starting in the late 1990's, a plethora of research projects were published and code was put into publicly accessible repositories. The researchers took two different approaches, Dual-Kernel and In-Kernel.

The Dual-Kernel approach uses a Nano- or Micro-Kernel as the RTOS kernel which runs Linux in its idle task. A similar approach has been used before to provide Real-Time extensions for MS-DOS, WINDOWS and UNIX variants and is still in use for commercial WINDOWS Real-Time extensions as of today. The Dual-Kernel

approach requires only minimal changes to the Linux kernel and and the Nano- or Micro-Kernel RTOS implementation is relatively straightforward.

The In-Kernel approach modifies the Linux kernel itself to provide Real-Time capabilities. Because the Linux kernel was not originally designed with Real-Time functionality in mind, it would require an enormous effort to achieve this. Determinism, being a system property, imposes restrictions and requirements on nearly every aspect of an operating system's functionality.

Nano- or Micro-Kernels can be formally verified as an isolated entity. However, ensuring that the Linux GPOS kernel, which runs with the RTOS kernel on the same processor system, cannot violate the correctness requirement of the RTOS kernel is a significant challenge.

Dual-Kernel systems provide a separate Application Programming Interface (API) for the Real-Time tasks and require specialized mechanisms to communicate and synchronize with the non Real-Time applications which run on the Linux kernel. Some of these approaches required the Real-Time application to be loaded as a kernel module, which not only caused licensing concerns, but also put severe limits on debuggability and the use of run-time analysis tools.

In-Kernel systems provide a uniform API for Real-Time and non Real-Time applications. However, there is a caveat: Real-Time applications have to carefully choose which system services to use, as many of them are non-deterministic. This restriction is similar to the restricted Real-Time API functionality in RTOSes and Dual-Kernels. Despite this, Real-Time applications can be debugged and exposed to run-time analysis tools just like any other application.

The well known and influential research projects from this time are RTLinux, RTAI, L4Linux, KURT and Linux/RK. RTLinux, RTAI and

L4Linux use the Dual-Kernel approach. KURT and Linux/RK aimed for In-Kernel Real-Time.

The debate between proponents of the Dual-Kernel and the In-Kernel approaches, which started in the 1990s, continues today and may never be fully resolved. In many aspects, it is like the well-known Micro-Kernel versus Monolithic Kernel debate, which unfolded 1992 between Prof. Andrew Tannenbaum and Linus Torvalds.

The early Linux Real-Time community

The first project, RTLinux, initially gained some traction in the open-source world. However, interest remained limited to a small group of enthusiasts due to the licensing restrictions that required applications to be licensed under GPL version 2.

Subsequently, RTAI attracted a wider community because, initially, it appeared to be free of patent encumbrances. This assumption, however, later turned out to be problematic. Though this situation was remedied by Xenomai, the adoption of RTAI stayed rather limited.

Neither the KURT nor the Linux/RK project succeeded in attracting an open source community around them due to the initial popularity of RTLinux and RTAI, which provided a more complete solution. However these projects had significant influence on the Linux kernel community based Real-Time efforts, as they successfully demonstrated the general viability of the In-Kernel approach.

These projects operated independently from the Linux kernel development, and none of them prioritized merging their changes into the mainline kernel. In 1999, Peter Wurmsdobler and Nicholas McGuire organized the first Real-Time Linux Workshop (RTLWS) in Vienna to bring together academics, open source developers, and industry engineers. This initial workshop inspired passionate

academic debates between the Dual-Kernel and In-Kernel proponents, while open source developers and industry representatives attempted to make sense of the discussions.

For many years, RTLWS was the premier Linux Real-Time conference, hosting 18 events. By 2006, the growing participation and interest from the Linux kernel community developers justified starting a dedicated kernel developer track, which was first introduced at the 2009 Dresden conference. This milestone laid the ground for ongoing, successful collaboration between the academic and Linux kernel communities, a partnership that continues to thrive. In 2018, RTLWS transformed into Real-Time tracks at the Embedded Linux Conference (ELC) and the Linux Plumbers Conference (LPC).

The start of Preempt-RT

In 2002, the Linux kernel community started to improve the responsiveness of the Linux kernel by introducing explicit preemption points. This allowed breaking up potentially long running sections of kernel code which caused large latencies. The user visible effect of this work was enhanced responsiveness of the Linux desktop. For server applications, this change reduced sporadic latencies, especially in network intensive workloads.

Various embedded Linux vendors tried to leverage these changes to provide basic Soft Real-Time capabilities to their customers. The changes were not sufficient for their customers‘ needs, so developers working for those vendors implemented improvements to address particular customer specific issues. Then, in September 2004, several developers working for those vendors posted patches with their changes. Since these changes were primarily developed to address specific needs, none offered a comprehensive Real-Time solution. This led to one of the largest email threads on the Linux kernel mailing list, highlighting the lack of community

consensus on the general approach to putting an In-Kernel solution in the Linux kernel upstream. Developers from Dual-Kernel projects argued that these changes were unnecessary because they already had a ready-to-use solution. This view was generally not well received by the kernel developers, whose priority was to improve the kernel itself. Unfortunately, parts of the email discussion derailed into futile arguments over minor differences measured in single digit microseconds.

Ingo Molnar, who was at the time the maintainer of the Linux kernel scheduler and involved in the low latency efforts, picked up some of the patches, or rewrote them from scratch. He combined them into a consistent set of patches aimed at achieving full In-Kernel Real-Time support. With this starting point, a small team of core developers was formed. Thomas Gleixner, who had worked with the late Douglas Niehaus on KURT/LibeRTOS, brought in technology from these efforts. Steven Rostedt, who was familiar with the Linux/RK work, brought in expertise and ideas. Various other developers joined the effort and helped to create the first usable Linux kernel community Real-Time prototype. This effort is since known as the ”Linux Real-Time preemption patch“, or more succinctly, ”Preempt-RT“, named after the related kernel configuration option.

From the very beginning, this group of developers aimed to integrate the Real-Time work into the upstream kernel. At the Linux Kernel Summit 2005 in Ottawa, the general approach of integrating Real-Time into the upstream kernel was discussed. Most developers agreed with the approach, with the stipulation that it did not disrupt existing work and create roadblocks for future development.

Aside from the Dual-Kernel proponents claiming that this was ”mission impossible”, the scope of the undertaking was not clearly defined at that point. In 2004, Ingo Molnar phrased it this way:

Are you sure that you want to touch every single file in the kernel while working at this for the next 10 years?

His question was considered hyperbolic at the time, but it turned out to be close to reality. In fact, no one was able to accurately estimate what it would ultimately take to convert the initial prototype into a maintainable solution which could be integrated into the upstream Linux kernel.

Aside from the undefined scope, there were no answers on how to solve the technical challenges. Solutions needed to be found that would not conflict with the performance and throughput goals of the Linux kernel community and keep pace with the ever changing code base of the Linux kernel. Moreover, no one had any idea how many unknown technical problems would be unearthed over time.

Twenty years of work

During the first five years, a substantial part of the Real-Time work found its way into the Linux kernel. Aside from generalization efforts to reduce redundancy in CPU architecture specific implementations, significant infrastructure work was contributed and accepted.

All of this work was required to make Preempt-RT feasible, but none of it was truly Preempt-RT specific. Generalizing code, adding infrastructure which helps to improve correctness, debuggability and maintainability, and adding features which are useful outside of the Real-Time realm has improved the quality and usefulness of the Linux kernel for everyone.

This period has a long list of achievements. Generic timekeeping, generic and threaded interrupt handling, high resolution timers, the mutex infrastructure, priority inheritance for user space mutexes, preemptible and hierarchical RCU, the tracing infrastructure,

the lock dependency validator, and several other advancements emerged from this effort.

After that period, the number of active developers decreased significantly, which caused the project to go mostly into maintenance mode and to keep it up to date. The required work to upstream the remaining pieces was too large to be handled by a few people with limited funding. In 2015, the funding situation was remedied and the upstream efforts resumed.

The Real-Time team at Linutronix focused on major refactoring tasks, which primarily improved the quality and stability of the upstream Linux kernel code base, but at the same time, laid the ground to integrate the Real-Time mechanisms smoothly. Gleixner and his team thereby strictly followed the advice from Linus Torvalds:

> *In other words, every new crazy feature should be hidden in a nice solid "Trojan Horse" gift: something that looks _obviously_ good at first sight.*

This effort reduced the size of the Preempt-RT patch set significantly and led to the more interesting and hard to solve Real-Time specific problems. While the scope of these problems were more localized than the already merged refactoring work, the remaining pieces were more difficult to integrate cleanly. They were also harder to justify for merging into the upstream kernel as they couldn't be hidden in a "Trojan Horse" gift anymore.

While most of the technical challenges could be addressed within the projected timelines, one surprising holdout kept the team under suspension for many years: printk.

printk is an infrastructure, which provides the kernel with the ability to emit informal and emergency messages on the consoles. The printk implementation in the kernel rooted back to Linux version

1.0 and its design never significantly changed, while developers tried to address its short-comings with creative workarounds for almost two decades.

While the informal part is trivial and non-critical, emergency messages expose an abundant heap of hard to solve technical challenges. These messages are emitted when the kernel encounters a bug or one of the runtime debug mechanisms detects a critical state. The contexts in which these messages can be emitted are completely arbitrary and include nesting into already ongoing output operations. As these messages are crucial for problem analysis, it is important to ensure that they reach the console output.

Addressing this required another round of large scale refactoring work. Handling the critical context requirements resulted in going back and forth between implementation and the drawing board several times. At LPC 2023, the latest approach was demonstrated and discussed with the kernel developers. Aside from implementation details, this solution is holding up so far and is on the way to being merged into the mainline kernel. Once this is completed, the configuration switch to select Preempt-RT in the mainline kernel can be enabled for the supported architectures.

How Real-Time is Preempt-RT really?

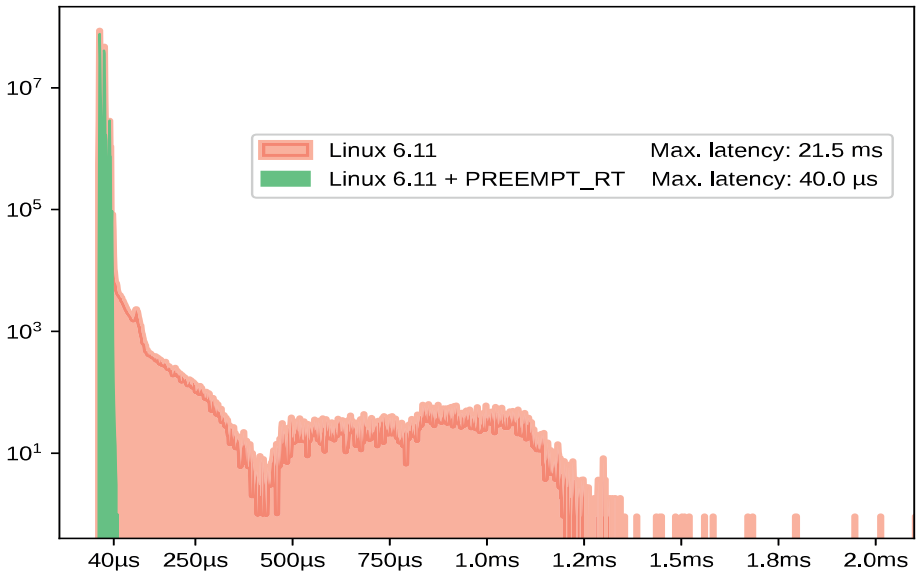
While Preempt-RT is demonstrably more advanced than Soft Real-Time, it lacks the formal proof to be correct under all circumstances. For many systems which do not require formal verification, the statistical proof of correct behavior is sufficient. With careful system design and tuning Preempt-RT is able to satisfy the requirements of the majority of Real-Time application requirements.

The lack of formal verification has been brought up against Preempt-RT for a long time, especially in the context of functional

safety. For those keeping track, the Dual-Kernel approaches are not truly verifiable either due to the fact that the two kernels share the same address space and malfunction of the GPOS kernel can definitely cause disruption of the RTOS. The In-Kernel approach suffers from the same problem. Additionally the temporal non-interference of all parts of the kernel would have to be verified.

As the In-Kernel approach of Preempt-RT has many advantages over a Dual-Kernel implementation and the verification of the combined Dual-Kernel is non-trivial, the industry started to research the feasibility of certifying Linux based systems on multicore CPUs.

The industrial research project SIL2LinuxMP was started in 2015 with the goal to determine whether it is possible to build complex software-intensive safety-related systems using the Linux operating system and Preempt-RT as its foundation. This research approached the safety requirements by leaving the well established



Comparison between RT and Non-RT kernel

route of compliant development and moving from a development to a controlled selection and analytical process. The project did not achieve a full safety argumentation, but laid the foundation for its successor ELISA (Enabling Linux In Safety-Critical Application). ELISA is a collaborative project under the umbrella of the Linux Foundation.

Both SIL2LinuxMP and ELISA have resulted in novel analysis tools, which are not only useful in the context of functional safety, but also for the general proof of correctness of the Linux kernel, which is a welcome contribution to further stabilize the code base for everyone.

Like the developers of Preempt-RT the safety community has gone off the beaten track and explored innovative concepts to solve these challenges. One outstanding approach worth mentioning is the Runtime Verification tool developed by the late Daniel Bristot de Oliveira.

Bristot turned the classic fine-grained model checking and theorem proving approach at instruction level into a method to validate the runtime behavior of a system against a formal specification of the expected system behavior. This approach is formally verifiable and can be applied to complex systems without requiring the Herculean task of re-implementing the complete system in a formal modeling language.

Whether Preempt-RT can be certified on its own is still an open question, but the efforts from these projects have brought more stability and boosted the confidence in the general correctness of the system.

Aside from that, the lack of verification does not preclude Linux/Preempt-RT from being used in mission critical systems. Careful overall system design, redundancy and diversity concepts make this possible. This is not a new and Preempt-RT specific approach.

Mission critical systems based on a combination of WINDOWS 95 and Linux 2.2 have been designed, certified and deployed in the late 1990s already.

Linux/Preempt-RT in the real world

Linux/Preempt-RT has been successfully deployed for more than a decade in a broad range of application spaces including telecommunications, medical devices, data acquisition, industrial automation, robotics, automotive, aerospace and satellite technologies. It is part of the offerings of many community and enterprise distributions and supported by most of the embedded Linux vendors.

The main reasons for the wide adoption are the broad hardware support, scalability, security and the seamless integration into the larger Linux ecosystem.

What‘s next?

One might assume that after enabling support for Preempt-RT in the mainline Linux kernel, the Real-Time challenge is over, but in reality there are still a lot of problems to be solved. The constant influx of new technologies and the ever increasing chase for performance and scalability optimizations in the Linux kernel will provide a source for new challenging problems to be solved for a very long time.

One outstanding problem to mention is in debate since 2009. At the RTLWS 2009, the late Doug Niehaus presented a novel concept to replace the well understood priority inheritance mechanism with a new concept of Proxy Execution. This solves the long standing practical and theoretical problem that priority inheritance applied to deadline and bandwidth scheduling policies does not work correctly and can lead to unbound latencies.

The proposed mechanism seems straightforward at first sight and can be readily implemented for single processor systems, but the reality of multiprocessor systems makes it a hard to solve puzzle. Several attempts to implement it failed over the years and it is still being worked on, but the most recent efforts look promising.

Linux/Preempt-RT in numbers

During the 20 years of development, about 10,000 patches related to Preempt-RT, which originated with the developers who were involved in the project over time, have been merged upstream. Some of the refactoring and generalization work resulted in follow up changes done by other developers who converted architectures or drivers to more modern interfaces. These direct secondary effects are hard to quantify. A rough estimate based on commit subjects puts them into the range of 5,000, but that number has to be taken with a grain of salt.

The largest efforts directly attributed to the project were the refactoring of CPU hotplug code, the removal of the Big Kernel Lock (BKL) and new infrastructures in the areas of timers, scheduling and locking.

The number of changes are taken from the upstream kernel Git history and therefore do not reflect the actual number of patches which were submitted to the mailing lists, nor the number of patches which were initially developed in the context of the project. Many of the topics have gone through several iterations in the Preempt-RT patch set before they were submitted for inclusion into the mainline kernel. They also do not take the number of subsequent developments into account, for it is not necessarily related to Real-Time. To put this into perspective, the RT-Mutex infrastructure was rewritten about five times in the early days of Preempt-RT before an attempt to include it upstream was made. The various attempts to solve the printk problem have accumulated easily

into more than 1,000 patches written over the course of six years. Most of the work has been done in the beginning by a core team of about ten developers. The final upstreaming effort was mostly handled by a smaller team of up to five developers with the help of the larger community. Of course this would not have been possible without the help of many other contributors. The effort to remove the BKL involved~80 developers, though only a few of them were engaged with or interested in Real-Time. Due to the blurry history of the out of tree patches, it's almost impossible to quantify the number of contributors correctly. From the inspection of mail archives, it is approximately 150 to 200.

Impact on the Linux kernel

The Preempt-RT project resulted in presumably the largest refactoring effort in the history of the Linux kernel touching a broad range of kernel subsystems. The Real-Time mechanisms required far stricter semantics than those originally provided by the Linux kernel. In the early days, some developers raised concerns that the introduction and enforcement of stricter semantics would restrict their freedom and creativity. While it is true that stricter semantics prevent some creative shortcuts, the benefits of enforcement, aided by run-time analysis tools, did not have a negative impact on the further evolution of the kernel. Quite the contrary, the enforcement allowed developers to validate their approaches for correctness, which resulted in a net increase of stability.

Linus Torvalds acknowledged this benefit in 2010:

> *But on the whole, I think it's actually worked out pretty well for them. I think the mainline kernel has improved in the process, but I also suspect that_their_RT patches have also improved thanks to having to make the work more palatable to people like me who don't care all that deeply about their particular flavor of crazy.*

Due to that process, nearly all implementation details of the Real-Time changes have evolved from ad-hoc workarounds to well-defined mechanisms. Many of them provide analysis benefits even when Preempt-RT is disabled.

Compared to twenty years ago, the size of the kernel grew from 4.2 million to 28.1 million lines of code. The vast majority of the growth is in drivers. In the early days of Preempt-RT, drivers were especially problematic for Real-Time. Most of these problems were hidden for a long time due to the lack of enforced semantics, tooling, clear interfaces and abstractions at the time when these drivers were implemented. These bugs were hard to trigger and not reproducible, which made them either silent errors or hard to analyze. The Real-Time modifications exposed such issues reliably, so the Real-Time developers spent a lot of time analyzing and addressing them.

The continuous improvement of abstractions and tooling improved the quality and stability significantly and the majority of drivers today work correctly on Real-Time enabled systems out of the box.

This is not solely an achievement of the Real-Time efforts. During the last twenty years, tooling, compilers, static code analysers and run-time validation mechanisms have improved significantly. Consolidation in driver subsystems and other areas of the kernel have contributed to this as well.

Educational efforts

The early concerns regarding restricting freedom and creativity had a surprising longevity. A long trail of discussions around the tradeoffs between enforced correctness, which aids stability, and the permanent quest to prioritize new features and performance can be found in the Linux kernel mailing list archives. Despite

the vast amount of publicly accessible information, this discussion seems to continue forever. But that's not only a Real-Time specific problem. In 2013 Dave Chinner, the XFS file system maintainer, said:

> *I've just given up trying to convince people to use the generic code when they are set on micro-optimising code. The "I can trim 3 instructions from every increment and decrement" argument seems to win every time over "we know the generic counters work"....*

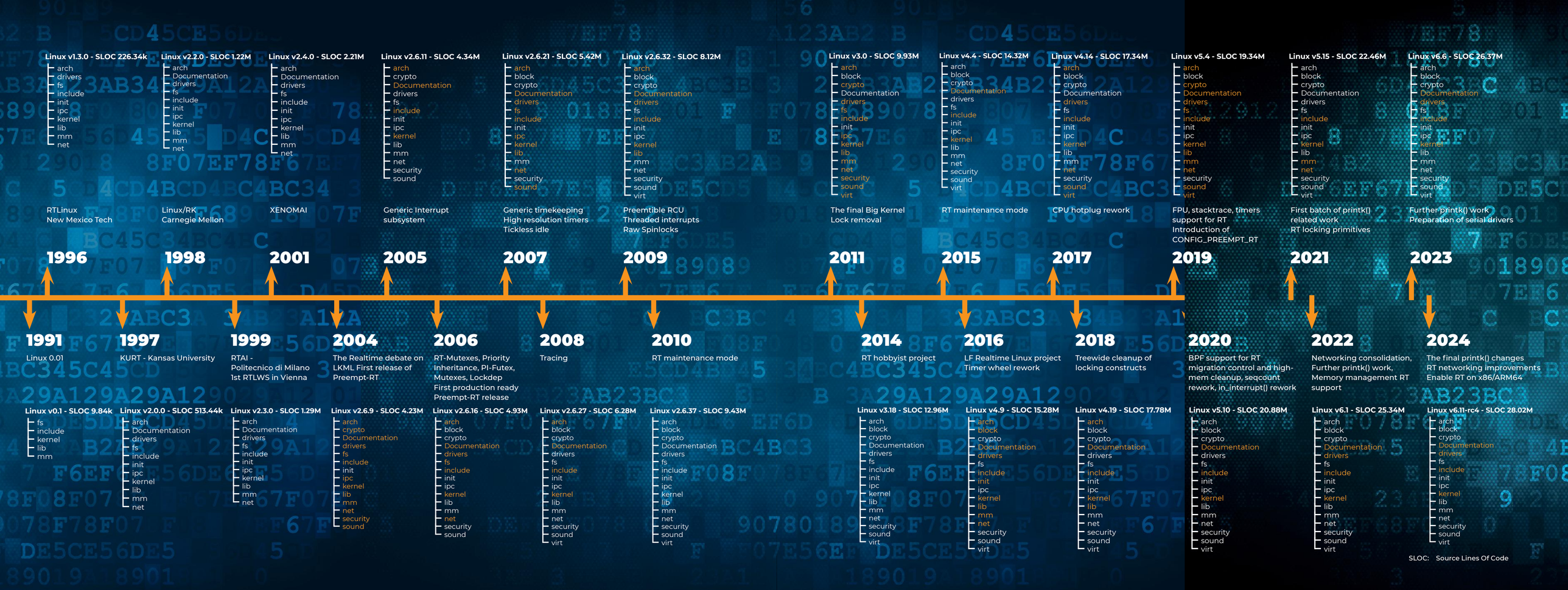
Seven years later, the approach of tooling enforced correctness was put into question by developers of a new subsystem. They tried to hand wave away the restrictions to instrument code which is de facto not instrumentable for technical reasons. Thomas Gleixner's answer to this was:

> *If we can have technical means to prevent the wreckage, then not using them for handwaving reasons is just violating the only sane engineering principle:*
>
> *Correctness first*
>
> *I spent the last 20 years mopping up the violations of this principle. We have to stop the "features first, performance first" and "good enough" mentality if we want to master the ever increasing complexity of hardware and software in the long run. From my experience of cleaning up stuff, I can tell you, that correctness first neither hurts performance nor does it prevent features, except those which are wrong to begin with.*

Is "Mission Impossible" really impossible?

Retrofitting the Linux kernel with Real-Time capabilities was considered impossible for various reasons. Aside from the theoretical concerns about the general concept, which will be discussed later, the main obstacles were:

Important steps of Preempt-RT


1991
Linux 0.01
Linux v0.1 - SLOC 9.84k
fs include kernel lib mm
1996
RTLinux
New Mexico Tech
Linux v1.3.0 - SLOC 226.34k
arch drivers fs include init ipc kernel lib mm net
1997
KURT - Kansas University
1998
Linux/RK
Carnegie Mellon
Linux v2.0.0 - SLOC 513.44k
arch Documentation drivers fs include init ipc kernel lib mm net
Linux v2.0 - SLOC 1.22M
arch Documentation drivers fs include init ipc kernel lib mm net
1999
RTAI -
Politecnico di Milano
1st RTLWS in Vienna
Linux v2.3.0 - SLOC 1.29M
arch Documentation drivers fs include init ipc kernel lib mm net
2001
XENOMAI
Linux v2.4.0 - SLOC 2.21M
arch Documentation drivers fs include init ipc kernel lib mm net
2004
The Realtime debate on
LKML First release of
Preempt-RT
Linux v2.6.9 - SLOC 4.23M
arch crypto Documentation drivers fs include init ipc kernel lib mm net security sound
2005
Generic Interrupt
subsystem
Linux v2.6.11 - SLOC 4.34M
arch crypto Documentation drivers fs include init ipc kernel lib mm net security sound
2006
RT-Mutexes, Priority
Inheritance, PI-Futex,
Mutexes, Lockdep
First production ready
Preempt-RT release
Linux v2.6.16 - SLOC 4.93M
arch block crypto Documentation drivers fs include init ipc kernel lib mm net security sound
2007
Generic timekeeping
High resolution timers
Tickless idle
Linux v2.6.21 - SLOC 5.42M
arch block crypto Documentation drivers fs include init ipc kernel lib mm net security sound
2008
Tracing
Linux v2.6.27 - SLOC 6.28M
arch block crypto Documentation drivers fs include init ipc kernel lib mm net security sound virt
2009
Preemtible RCU
Threaded interrupts
Raw Spinlocks
Linux v2.6.32 - SLOC 8.12M
arch block crypto Documentation drivers fs include init ipc kernel lib mm net security sound virt
2010
RT maintenance mode
Linux v2.6.37 - SLOC 9.43M
arch block crypto Documentation drivers fs include init ipc kernel lib mm net security sound virt
2011
The final Big Kernel
Lock removal
Linux v3.0 - SLOC 9.93M
arch block crypto Documentation drivers fs include init ipc kernel lib mm net security sound virt
2014
RT hobbyist project
Linux v3.18 - SLOC 12.96M
arch block crypto Documentation drivers fs include init ipc kernel lib mm net security sound virt
2015
RT maintenance mode
Linux v4.4 - SLOC 14.32M
arch block crypto Documentation drivers fs include init ipc kernel lib mm net security sound virt
2016
LF Realtime Linux project
Timer wheel rework
Linux v4.9 - SLOC 15.28M
arch block crypto Documentation drivers fs include init ipc kernel lib mm net security sound virt
2017
CPU hotplug rework
Linux v4.14 - SLOC 17.34M
arch block crypto Documentation drivers fs include init ipc kernel lib mm net security sound virt
2018
Treewide cleanup of
locking constructs
Linux v4.19 - SLOC 17.78M
arch block crypto Documentation drivers fs include init ipc kernel lib mm net security sound virt
2019
FPU, stacktrace, timers
support for RT
Introduction of
CONFIG_PREEMPT_RT
Linux v5.4 - SLOC 19.34M
arch block crypto Documentation drivers fs include init ipc kernel lib mm net security sound virt
2020
BPF support for RT
migration control and high-
mem cleanup, seqcount
rework, in_interrupt() rework
Linux v5.10 - SLOC 20.88M
arch block crypto Documentation drivers fs include init ipc kernel lib mm net security sound virt
2021
First batch of printk()
related work
RT locking primitives
Linux v5.15 - SLOC 22.46M
arch block crypto Documentation drivers fs include init ipc kernel lib mm net security sound virt
2022
Networking consolidation,
Further printk() work,
Memory management RT
support
Linux v6.1 - SLOC 25.34M
arch block crypto Documentation drivers fs include init ipc kernel lib mm net security sound virt
2023
Further printk() work
Preparation of serial drivers
Linux v6.6 - SLOC 26.37M
arch block crypto Documentation drivers fs include init ipc kernel lib mm net security sound virt
2024
The final printk() changes
RT networking improvements
Enable RT on x86/ARM64
Linux v6.11-rc4 - SLOC 28.02M
arch block crypto Documentation drivers fs include init ipc kernel lib mm net security sound virt
SLOC: Source Lines Of Code



1. The unknown scope of the overall effort
2. Working against a continuously changing codebase
3. The lack of a dedicated team of experts over a longer period of time

The decision to pursue the project despite the unknown scope of the overall effort was based on the earlier results from related academic research projects and the initial proof of concept implementation. These results demonstrated that the concept was feasible. Assessing the over-all effort was impossible, but considered a manageable risk if an appropriate engineering approach was found.

The rate of changes in the Linux kernel is enormous. About 1.2 million changes have been applied to the Linux kernel in the past twenty years. That's on average ~60,000 changes per year, ~165 changes per calendar day or ~6.9 changes per hour. The rate per hour increased by a factor of ~2.7 over that time period. The average growth rate of the Linux kernel code base in that timeframe is about 10% per year.

The average of ~2.7 changes per hour in 2004 and a 10% growth rate might have been reason enough to abandon the project and resort to the Dual-Kernel approach or start a dedicated Real-Time project from scratch. Neither one of these alternatives was considered attractive. The Dual-Kernel approach would have lost the advantage of the consolidated programming interface and faced strong opposition for clean integration as expressed by various kernel maintainers. The main objection was that the inclusion of the necessary Real-Time hooks would not benefit the kernel itself, but add additional maintenance costs. Starting a new Real-Time kernel project from scratch was not a viable option either. The effort to reimplement a kernel on par with the Linux kernel was estimated to require about 4,500 man years in a study published by David A. Wheeler in 2004.

Contrary to that, the In-Kernel approach allowed leveraging the broad and increasing hardware support, infrastructure like the network stack and the ongoing scalability efforts. This was the decisive factor to pursue the goal of bringing Real-Time capabilities to the Linux kernel. Again, the assumption was that the change and growth rate are manageable, if an appropriate engineering approach could be determined.

The lack of a dedicated team of experts committed to the project over a longer period of time was considered as well. Though it seemed irrelevant at the time, because the initial interest and participation did not indicate that this would become an issue.

So the developers accepted the challenge in the spirit of Nelson Mandela:

> *It always seems impossible until it's done.*

Pragmatic evolution

If the Real-Time developers would have started with a full design specification for Preempt-RT, the project would not even have reached the prototype stage by now. The developers chose the approach of pragmatic evolution instead. Pragmatic evolution continuously refines both the engineering process and the implementation in a rigorous feedback cycle.

The process started with systematic analysis to provide a fine grained inventory of issues which needed to be addressed. While many of these issues were interrelated, the breakdown allowed the work to be split up into manageable subtasks. This enabled engineers to work in parallel and integrate the results continuously. The analysis had to be refined regularly to take the accomplished work and the concurrently ongoing evolution of the upstream kernel into account.

The ability to refine the process and the course taken turned out to be beneficial right from the start. The Real-Time changes resulted in a different run-time behavior of the Linux kernel exposing a rather large amount of latent bugs in the existing kernel code base. A latent bug is an existing error that has yet not caused a failure because the exact condition to trigger it was never fulfilled. The effort required to analyze and address these bugs was beyond the capacity of the developer team. This unexpected outcome was addressed by resorting to tooling and run-time analysis, which became another crucial part of the overall approach.

Adding tooling and run-time analysis to expose latent problems during development and testing into the upstream kernel was mostly uncontroversial because it provided an immediate benefit, independent of the Real-Time effort. The upstream integration of these tools reduced the efforts for the Real-Time developers to analyze and address the problems significantly. Addressing the problems found by the tools shifted to the upstream developers and the relevant experts of the affected code. With the larger exposure in the upstream kernel, the tools had to be refined and adjusted, but all of the tools became indispensable for kernel development.

Keeping up with the concurrent changes and the continuous growth of the upstream kernel code base turned out to be less problematic than it looked at first sight. One significant aspect is that the tools identify problematic changes for the most part, before they reach the upstream kernel. The other aspect is early integration. The fine grained subtasks were refined over time in the out of tree Preempt-RT patch. The regular releases of the patch set provided sufficient testing to gain confidence in the combined outcome. Functional changes were systematically separated from preparatory changes. Preparatory changes included consolidation of duplicated code, encapsulation of code constructs into better

abstractions and interfaces. These preparatory changes did not change the functionality or the behavior of the upstream kernel. After they reached the point of stabilization, these cleanups were integrated upstream.

The upstreaming of the cleanups ensured that new code used the new mechanisms. This allowed the Real-time developers to focus on modifying these new abstractions and interfaces instead of chasing problematic code constructs across the code base.

A similar approach was taken for infrastructure replacements. The new infrastructure was carefully designed to utilize the required parts of the existing infrastructure first. This allowed the introduction of a new control mechanism without changing any other code. Once the new infrastructure was in place, the code utilizing the original infrastructure was converted over to the new mechanism step by step. After the conversion of the codebase was complete, the obsolete parts of the original infrastructure were removed.

Over the years the process was continuously refined and expanded by reflection, adopting new tools and learning from other efforts. The design and implementation details went through corresponding refinement cycles. New ideas were prototyped to study the feasibility. Revisiting the drawing board more than once was a regular consequence of these studies. Once the design settled several rounds of refinements were required, first in the Real-Time tree, and then again on upstream submission.

The engineering approach of pragmatic evolution was successful, but not sufficient on its own. It required a great deal of perseverance along with it as Albert Einstein put it aptly:

> *It‘s not that I‘m so smart, it‘s just that I stay with problems longer.*

Criticism

From the outset, the Preempt-RT project was met with criticism from parts of the academic Real-Time research community, because it pragmatically sets aside established academic theories in favor of exploring solutions that held up in real-world scenarios. This was necessary as many of the Real-Time theories were established in idealized ”clean room” environments, and did not account for the reality of complex multiprocessor systems and the requirements of today’s Real-Time applications.

Victor Yodaiken, the creator of the Dual-Kernel RTLinux project, criticized the Preempt-RT project by saying:

> *My opinion has always been that the Linux-RT project was based on an unfixable engineering error.*

It‘s unclear whether the ”unfixable error” is the monolithic design of the Linux kernel itself or the pragmatic approach to violate established Real-Time theories. Nevertheless, this so-called unfixable engineering error powers the most demanding Real-Time applications in the world.

Acknowledgements

The Linux Real-Time community and the Linutronix Real-Time team are deeply grateful to everyone who contributed patches, ideas, bug reports, testing and documentation to this effort.

We extend our thanks for the support we received in the past twenty years from developers, companies and organizations that supported and funded this incredible journey.

The Linux Real-Time community has thrived for 25 years and we eagerly anticipate celebrating the 50-year anniversary!

Retrospectives

My Linux RT adventure

Roberto Bucher, June 26, 2024

In the year 2000, I found myself in a rather unusual situation. After a nasty motorcycle accident, I was stuck in bed for what felt like an eternity. Bored out of my mind and tired of watching daytime TV, I decided to dive into something new and exciting. That's when I stumbled upon the fascinating world of Linux Real-Time (RT) variants.

With plenty of time on my hands, I dug into the details of RT Linux and Linux RTAI. After much pondering and a bit of a coin flip, I chose Linux RTAI. Why? Well, it didn't hurt that Milan, where RTAI was being developed, was just ahop, skip, and a jump away from my home in Lugano.

My first big project? Creating a kernel module from Matlab/Simulink. I remember the thrill of typing "insmod" and watching my creation spring to life in the kernel space. It was like magic, only with more code and fewer rabbits.

In 2003, I hit the jackpot when SUPSI offered me a sabbatical semester in Milan. I joined Paolo Mantegazza's team, and suddenly I was like a kid in a candy store, but instead of sweets, I was surrounded by code and brilliant minds. My mission? To work on code generation for Linux RTAI (RTAILab), using block diagrams from Scilab/Scicos. It was like building intricate Lego structures, only way cooler.

For the next few years, I lived and breathed RTAI. My colleagues joked that I should have named my dogs "RTAI" and "Scilab" given how often I mentioned them. But then came 2006, and with it, the grand unveiling of Preempt-RT at the RTWLS in Lanzhou. It was like being hit by a lightning bolt of innovation. The room

buzzed with excitement, and I could almost hear the collective ”wow” as the possibilities unfolded before us.

That moment was a game-changer. Preempt-RT was poised to revolutionize everything we knew about real-time Linux systems. From then on, I shifted my focus to automatic code generation, eventually developing my own tool under Python, called pysimCoder. My journey with Linux RT had been a wild ride from convalescence to cutting edge developments, filled with code, camaraderie, and a fair share of geeky excitement. And so, the adventure continues, one line of code at a time.

My journey towards Real-Time Linux

Peter Wurmsdobler, July 2024

While I was working on my PhD in control theory, MATLAB used to be the preferred tool for all my numerical issues; I even did my accounting in MATLAB. At that time I merely had a peak into C and proclaimed: I will never ever write software in such a language; I prefer the pristine world of simulation where deterministic timing of discrete time control systems is not a worry and always perfect. And if not, then The Mathwork’s Real-Time Workshop will oblige, or Nicholas McGuire who would implement my algorithms on a DSP. Soon I had to revisit my position.

After my PhD I moved to France where I had to build a machine controlled by a computer running MS-DOS, an Intel 486 based PC with a DAQ card. The machine’s system design was not an issue, nor the mechanical or the electronic design, but the control software was. Since there was no budget for The Mathwork’s fancy tools, needed to found an alternative solution. Nicholas helped me exploring options, Linux, in particular Real-Time Linux; so I joined the RTL mailing list where I found people to be very helpful. Paolo Montegazza for instance engaged in a productive discussion on the software design using real-time threads, interrupt handlers

and what not. At some point, however, he said: Peter, I think now it is time to stop talking and start getting your hands into C programming.

So I went to Paris, bought a book on C-programming, as well as books on operating systems, then installed SuSE and tried to understand more about Linux. To further my understanding, I travelled to Basel to see Tomek Motylewski and received a crash course on kernels, interrupt handlers and all the gory details of the PC architecture. After that I was ready to write some C programmes and kernel modules for my DAQ card; following a work visit from Tomek, I had a working solution with a user space program, a kernel module with a couple of RTLinux threads, some shared memory and an interrupt handler to churn out and record digital signals at a few kHz. this was the result of cooperation, support from a community and Nicholas (who remained my lifeline all along).

For me, having become a C programmer despite my initial reticence, there was only one snag: if my product depends on a favour of real-time Linux, I need some sort of guarantee that the interface remains stable and supported as well as its performance deterministic. Yet, on the RTL mailing list there was so much division, and discussion about the details which I neither understood nor cared much about. Therefore, Nicholas and I started talking about the urgent need for some kind of gathering that brings people together to reach some consensus and to define some stable programming interface. The only problem was some kind of reservation in participation by potential candidates from the mailing list upon asking them in direct messages about their interest.

An incentive was needed and some seed money. I borrowed $10k from my father and offered some compensation for participant's travel expenditure on a first-come-first-served basis. With a few people having taken the bait, names started to appear

on the attendee's list of the soon created web site; hence more participants signed up for the First Real-Time Linux Workshop to be held in Vienna in 1999. Nicholas later got the money back through a government grant as his alter ego, Der.Herr@Hofr.at, a risky nod to Austrian's imperial past, showing Austrian's propensity for military, academic or aristocratic titles). We could repay my father in full after a successful seminal workshop.

With all the support received from the real-time Linux community, I felt I would like to give something back. Since I was not able or capable to contribute much code to the kernel development myself, I have tried to help by organising workshops for a few years, the Real-Time Linux Workshop. I do hope that these workshops had an effect:

First, it was lovely to meet people from the mailing list in person. Second, meeting in person helps deepening the sense of community and making the building of consensus easier. Last but not least, these workshops were at the beginning of the wider free and open source movement, the revenge of the nerds.

LWN.net's view on the Real-Time history

Jonathan Corbet, August 2024

The Linux kernel first made an appearance in late 1991; interest in real-time uses of Linux followed not long after. By the time that LWN was just getting off the ground in early 1998, the first RT-Linux patch set was aiming for a stable release.

Decades later, we might just be about to finish the job of providing real-time response in a stock Linux system.

The early developers working on real-time Linux (prior to the real-time preemption work) were not the most agreeable folks, but they all seemed to agree on one fundamental point: they did not believe that Linux itself would ever be able to operate as a

real-time kernel. So the approaches favored at that time worked by pushing Linux out of the way, displacing it as the true kernel of the system and, instead, running it as a low-priority process under a minimal, non-Linux kernel. RTLinux took that approach (and even patented it), and RTAI followed with a variation on it, and one of the classic early Linux flame wars was launched.

When the first Real-Time Linux Workshop was announced to be held in Vienna in December 1999 , an explicit goal was to get the RTLinux and RTAI developers to treat each other as human beings and cooperate toward a useful real-time solution. Optimism never dies, it seems.

Some years went by without a lot of apparent progress, but much was happening behind the scenes. By late 2004, developers were increasingly tired of the RTLinux and RTAI camps and becoming more interested in incorporating real-time capability directly into Linux itself.

Thus there was a real-time patch set from Sven-Thorsten Dietrich, preemptible mutexes from Arnd C. Heursch, Dirk Grambow, Dirk Roedel, and Helmut Rzehak, the real-time security module from Torben Hohn and Jack O'Quin, the mmlinux work from Bill Huey, and a number of other initiatives.

It was unclear which, if any, of these initiatives would succeed, but the question was soon rendered moot. As often happened in that era, a much better implementation of the ideas contained in those patch sets emerged fully formed from Ingo Molnar's head after two days of meditation. Thus, on October 11, 2004, the real-time preemption patch set first hit the mailing lists. Ingo intended for this work to go upstream from the beginning:

> *I believe the basic concept is sound and inclusion is manageable and desirable.*

Optimism truly never dies. But, in truth, (almost all of) this work has found its way into the mainline kernel. The fact that it has taken two decades is a reflection of how difficult that task really was.

The difficulty here had two distinct aspects. One is the sheer challenge of modifying a general-purpose kernel with tens of millions of lines of code to be able to provide deterministic response times; there were many hard problems that had to be solved to get to that point. The other challenge was just as daunting, though: this work had to be integrated into the kernel in a way that did not degrade its operation for non-real-time workloads. Indeed, to justify the extra complexity that the real-time work brought, it often had to make things better for all workloads, despite the inherent trade-offs that must be made when prioritizing response time or throughput.

History shows that many developers give up when faced with that sort of task, but that is not what happened here. Over years, the real-time work did indeed make Linux better for all users. It brought us the lockdep locking checker, a reworked timer subsystem, robust futexes, the generic interrupt-handling layer, priority inheritance, dynamic tick support, the deadline scheduler, the lockless slab allocator, a reworked CPU hotplug subsystem, the runtime verification tools, a more robust console-logging subsystem, and far more. Each of these features has furthered the real-time project, but also improved the kernel as a whole. As Linus Torvalds described this process:

> *The RT people have actually been pretty good at slipping their stuff in, in small increments, and always with good reasons for why they aren't crazy.*

Now, 20 years after it began, the real-time preemption work might actually be reaching its conclusion. It has already been deployed in countless systems, so the merging of the final patches will be a nearly anticlimactic ending. This state of affairs is a testament

to two decades of focused, determined work from developers who were just as opinionated as the early real-time folks, but who were able to work productively with the rest of the community anyway.

Back in 2005, Ingo, perhaps foreseeing this outcome, suggested that the ending of the real-time preemption work would look like this:

> *So I'm afraid nothing radical will happen anywhere. Maybe we can have one final flamewar-party in the end when the .config options are about to be added, just for nostalgia, ok?*

RTLinux retrospective

Victor Yodaiken, Austin Texas, July 4 2014, vy1@e27182.com

RTLinux [3, 1, 5, 4] was a minimal real-time kernel that ran Linux as fully preemptible thread. Essentially Linux was the idle task for the real-time kernel. Later variants offered a BSD option in place of Linux. The system was intended for hard real-time with non-negotiable timing requirements.

A typical application involved a number of realtime threads collecting data and controlling a device such as a robot or instrument system, streaming data to Linux processes that would analyze, store, and display the data, and accepting control parameters back from the non-real-time code. Communication between real-time and non-real-time environments used shared memory and specialized pipes that were non-blocking on the realtime end.

Hard real-time capability allowed for a wide range of applications including security[6].

The original design was for systems with one or a few cores and the system relied on a novel lightweight virtualization of Linux interrupt controls[2] to keep Linux from delaying real-time code.

On 1990s commodity computers worst case latency was in the low microseconds. Later versions as cores became more available used a form of processor reservation and had significantly better latency.

The real-time kernel was optimized to limit worst case latency, particularly for thread scheduling, interrupts, and I/O. General purpose operating systems get many performance advantages from optimizing the common case. The dual kernel design allowed each kernel to be optimized for its most important performance metric. Uni-kernel real-time systems require an extensive and ongoing effort to compromise these two goals, to limit lock intervals and to make sure real-time tasks can't be blocked for too long by non-real-time code.

RTLinux was originally an open source project. A commercial version was added in the early 2000s and was sold to WindRiver Systems in 2007. At the time of the sale, we expected (incorrectly) that that multi-core computers made it too easy to produce competing systems using some variant of processor reservation for the real-time kernel.

References

[1] BARABANOV, M., and YODAIKEN, V. Real-time linux. Linux journal (February 1997).
[2] YODAIKEN, V. Adding real-time support to general purpose operating systems. US Patents and Trademarks Office, US-5995745-A, November, 1999.
[3] YODAIKEN, V. Cheap operating systems research. In Proceedings of the First Conference on Freely Redistributable Systems (Cambridge MA, February 1996).
[4] YODAIKEN, V. The rtlinux manifesto. In Proceedings of the 5th Annual Linux Expo (Raleigh, North Carolina, May 1999), pp. 187–197.
[5] YODAIKEN, V., and BARABANOV, M. Real-time linux. In Invited Talks: USENIX Winter Technical Conference (Anaheim, CA, Jan. 1997).
[6] YODAIKEN, V., and DOUGAN, C. Active semantically aware hard real-time security hypervisors. In Proceedings of the 4th Annual Workshop on Cyber Security and Information Intelligence Research: Developing Strategies to Meet the Cyber Security and Information Intelligence Challenges Ahead (New York, NY, USA, 2008), CSIIRW '08, Association for Computing Machinery.

Real-Time quotes

Linus Torvalds, 2002

Use a microkernel for the realtime stuff and be done with it!

Linus Torvalds, 2004

Real-time people are totally crazy!

Doug Niehaus, 2004

Real-time is not as fast as possible.
Real-time is as fast as specified.

Linus Torvalds, 2005

Friends, don't let friends use priority inheritance!

Steven Rostedt, May 2005

This really boils down to the terminology of hard and soft. Because, what I think of soft-RT is not as good as what the preempt-RT patch does. You need more too it. Probably, what I was talking about is diamond hard, and Ingo's RT patch is metal hard. PREEMPT is just wood hard and not PREEMPT is plastic hard. Leaving MS WINDOWS as feather hard.

↳ **Bill Huey**

Notating it in terms of Tofu firmness would have been more comforting.

↳ **Steven Rostedt**

Actually, since my wife is Italian, I should have used the hardness of spaghetti as it cooks. That way I could call MS WINDOWS an over cooked noodle!

Linus Torvalds, 2006

Controlling a laser with Linux is crazy, but everyone in this room is crazy in his own way. So if you want to use Linux to control an industrial welding laser, I have no problem with your using Preempt-RT.

Jonathan Corbet, Jan. 2008

The merging of the realtime Linux tree will be substantially complete by the end of the year.

↳ **Jonathan Corbet, Dec. 2008**

Your editor should know by now that expecting deterministic merge times for realtime patches is a sure path to disappointment; latencies in this area are always higher than one would like.

Jonathan Corbet, Jan. 2009

The realtime patch set will be mostly merged by the end of the year. It really has to happen this time. What could possibly go wrong?

↳ **Jonathan Corbet, Sep. 2009**

Significantly, nobody questioned the overall value of merging the realtime code into the mainline. Instead, some of the other discussions have made it clear that there are a lot of users for this functionality and that it is needed. So this merger will eventually happen, but your editor has learned better than to try to predict when.

Linus Torvalds, 2010

And yeah, I still think the hard-RT people are mostly crazy.

Jake Edge, Apr. 2011

It is obvious that Gleixner is tired of being asked for a roadmap for the realtime patches. Typically it isn't engineers working on devices or other parts of the kernel who ask for it, but is, instead, their managers who are looking for such a thing. There are several reasons why there is no roadmap, starting with the fact that kernel developers don't use PowerPoint. More seriously, though, the realtime developers are making their own road into the kernel, so they are looking for a road to follow themselves. But, so that it could no longer be said that he hadn't shown a roadmap, Gleixner presented one (see to the right) to much laughter.



Thomas Gleixner, 2014

The most intriguing idea so far was to jump on the momentum of the most hilarious crowd funding nonsense:

https://www.kickstarter.com/projects/324283889/potato-salad

So in consequence we might turn RT into a crowdfunded nonsense project which serves the purpose of controlling the potato-salad machine to make sure that Zack Danger Brown can deliver all the potato salad people have pledged for.

Jake Edge, Oct. 2017

Gleixner has presented various roadmaps for the realtime patch set over the years. This year's edition was textual: "Due to the evolutionary nature of Linux the roadmap will be published after the fact, but it will be a very precise roadmap." As with the others, this roadmap gives some insight into Gleixner's feelings about regularly being asked to provide one.

Jake Edge, Oct. 2017

For many years, Gleixner worked on the project as something of a hobby, but it is "much more fun to get paid for things", he said.

Peter Zijlstra, 2020

Right, so I'm concerned. migrate_disable() wrecks pretty much all Real-Time scheduler theory we have, and PREEMPT_RT bringing it in is somewhat ironic.

↳ **Thomas Gleixner**

It's even more ironic that the approach of Preempt-RT has been ‚pragmatic ignorance of theory' from the very beginning and despite violating all theories it still works.

↳ **Linus Torvalds:**

So either throw the broken theory away, or live with it. Theory that doesn't match reality isn't theory, it's religion. There are few things more futile than railing against reality, Peter.

↳ **Peter Zijlstra:**

But, but, my windmills! :-)

↳ **Thomas Gleixner:**

At least you have windmills where you live so you can pull off the real Don Quixote while other people have to find substitutes :)

The RTLWS archives

The archive of the Real-Time Linux Workshop proceedings is a rich source of information about the history of Real-Time Linux and the various approaches taken. The full archive is available at https://archive.kernel.org/rtlws-archive/

The following excerpts give some insight in to the history and hopefully inspire the interested reader to dig into the archives.

NMT-RTL

Michael Barabanov - RTLWS 1999

RTLinux is the hard realtime variant of Linux that makes it possible to control robots, data acquisition systems, manufacturing plants, and other time-sensitive instruments and machines. Version 1 of RTLinux was designed to run on low end x86 based computers and provided a spartan API and programming environment. Version 2 RTLinux is a complete rewrite, designed support symmetric multiprocessing, to run on a larger range of systems, and with extensions for ease of use.

In this paper, we will discuss the new system and its API with particular attention to the problems of increasing ease of use and adherence to standards, without performance compromise.

RED-Linux project

Yu-Chung Wang – RTLWS 1999

As the RED-Linux project is still in a very early stage, it is difficult for us to predict what will happen next. In a way it all depends on how much demand and help we can get from people like you. Moreover, it is our hope that many of you will find this project to be meaningful and interesting enough for your active participation. Your suggestion and comment will be highly appreciated.

DIAPM-RTAI for Linux: Whys, Whats and Hows

Paolo Mantegazza - RTLWS 1999

- NMT-RTL patch confirmed that 2.0.xx was not mature for RTHAI/RTAI;
- its simple scheduler, declared as primitive by NMT-RTL developers, was instead immediately recognized as what we needed, because it was very close to that of DIAPM-RTOS;
- So we could easily go to ”the old loved DOS way” and easily port all what we had under DOS (DI APM-RTOS almost unchanged, TSRs became LINUX modules);
- But the first tests were a disaster
- That’s why the DIAPM-RTAI variant was born

RT Linux works at the finest temporal granularity (1 microsec), but places RT computations in the context of the lowest level executive, not as part of Linux. KURT provides coarser time granularity (10s micro-sec), and is subject to scheduling distortions (10s micro-sec), but places the real-time computations in the context of Linux; both kernel and user modes. Linux/RK deals more with the ”resource kernel“ interface for describing resource sets and a locating their use to user level computations. The papers on Linux/RK claim roughly the same temporal granularity as KURT, but only report experiments with granularity at the 100s of milli-sec level.

Linux/RK - The resource kernel

Ragunathan (Raj) Rajkumar - RTLWS 1999

A resource kernel is defined to be one which provides timely, guaranteed and protected access to system resources. The resource kernel allows applications to specify only their resource demands leaving the kernel to satisfy those demands using hidden resource management schemes. This separation of resource specification from resource management allows OS-subsystem-

specific customization by extending, optimizing or even replacing resource management schemes. As a result, this resource-centric approach can be implemented with any of several different resource management schemes. The resource kernel gets its name from its resource-centricity and its ability to

- apply a uniform resource model for dynamic sharing of differenresource types,
- take resource usage specifications from applications,
- guarantee resource allocations at admission time,
- schedule contending activities on a resource based on a well-defined scheme, and
- ensure timeliness by dynamically monitoring and enforcing actual resource usage.

In summary, a resource kernel provides resource-centric services which, in turn, can be used to satisfy end-to-end QoS requirements. Generally, a QoS manager sitting on top of a resource kernel can make adaptive adjustments to resources allocated to applications.

RT-Mach is a resource kernel. See our recent work in the recent publications section on Processor Reservation and Disk Reservation.

The XENOMAI Project - Implementing a RTOS emulation framework on GNU/Linux

Philippe Gerum - RTLWS 2001

Xenomai is a GNU/Linux-based framework which aims at being a foundation for a set of legacy RTOS API emulators running on top of a host software architecture, such as RTAI when hard real-time support is required. Generally speaking, this project aims at helping application designers relying on legacy RTOS to move as smoothly as possible to a GNU/Linux-based execution environment, without having to rewrite their applications entirely.

This paper discusses the motivations for proposing this framework, the general observations concerning the legacy RTOS directing this project, and some in-depth details about its undergoing implementation.

RTLinux with address spaces

Frank Mehnert, Michael Hohmuth, Sebastian Schonberg,
Hermann Härtig – RTLWS 2001

In this paper, we determine the cost of a separate-space system relative to that of a shared-space system. We compare these particular two types of systems because we feel that separate-space systems lend themselves for the largest number applications, and we expect the largest performance over-head relative to shared-space systems.

For our evaluation, we used RTLinux from the shared-space systems category, and L4RTL, a reimplementation of the RTLinux API as a separate-space system based on a real-time microkernel and a user-level Linux server. We observed RTLinux‘s worst-case response time to be much higher than about 15 us on a generic x86 PC claimed by the systems authors. In our experiments, RTLinux worst-case interrupt latency was 68 us. The worst case for L4RTL was 85 us.

We found that the cost induced by address-space switches to real-time applications does not significantly distort the predictability of the system. In general, most of the worst-case overhead we observed must be attributed to implementation artifacts of the microkernel we used, not to the use of address spaces.

Use of cookies in Real-Time system development

M. Gleixner, M. M Guire – RTLWS 2009

While typical scientific works have focused on the technical aspects of real-time development and real-time systems this paper will focus on the caloric requirements that profoundly can impact development and notably scientific dissemination. Though the use of cookies and respective protocols in computer science are well documented we will not cover security aspects, notably related to excessive accumulative effects of consuming large amounts of cookies, rather we will focus on their creation, deployment, assessment and finally their consumption and the positive impact on the real-time Linux community we were able to observe.

Obituaries

Doug Niehaus



Doug Niehaus was an Associate Professor of computer science at Kansas University. He unexpectedly passed away in 2012 at the age of 54.

Doug has to be considered one of the pioneers of real-time Linux. His efforts of making Linux a venerable choice for real-time systems reach back into the mid 1990s. While his KURT (Kansas University Real-Time) project did not attract a large community, his influence on the Linux kernel and the Linux Real-Time Preempt-RT project reaches much farther than most people are aware of.

His research provided the initial proof that the Linux kernel could be modified to provide Real-Time properties laid the ground for two decades of work in the kernel community. His input on topics like high resolution timers, kernel instrumentation and scheduling has left its traces in the design of solutions which have found their way into the Linux kernel. His proposal to create a scheduling policy agnostic mechanism to prevent priorty inversion is still considered to be the correct approach, but it left developers and researchers puzzled over the implementation details for 15 years. Doug's direct and indirect influence on the Linux (Real-time) development is definitely worth to be mentioned and preserved in the Linux history book.

Doug's dry sense of humour, his language skills and his broad interest outside of computer science ranging from philosophy,

fine arts, music and literature to more prosaic topics like cooking, dogs and hiking were always a guarantee for lively and interesting conversations, which could get opinionated and heated as well. His open-minded, deeply humanistic, but also vulnerable character, which made it a pleasure and from time to time laborious to work and communicate with him.

The academic and the Linux Real-Time communities lost a brilliant mind and a friend, but his work left traces, which will be with us for ever.

Daniel Bristot de Oliviera



Daniel was a computer scientist with a focus on Real-Time systems and scheduling theory who is well recognized in the academic and the Linux kernel community. Daniel died prematurely at the age of 37 in June 2024.

His truly outstanding ability to apply theoretical Real-Time concepts to real-world problems in the industry has been instrumental in driving the success of Linux and its adoption in real-time critical application spaces.

Daniel was creative and passionate about computer science. He earned a joint PhD from Universidade Federal de Santa Catarina in Brazil and Scuola Superiore Sant'Anna in Italy, with a research thesis focusing on Automata-based Formal Analysis and Verification of the Real-Time Linux Kernel. His work was an exemplary piece of research, combining theoretical research arguments with a real implementation of a kernel-level mechanism. It models the behavior of complex parts of the Linux kernel, such as the process scheduler, with a finite-state machine and uses minimum-overhead run-time verification to validate the coherence of the kernel's run-time behavior and the theoretical model.

While he pursued his ideas and visions with great perseverance, he was always open for discussion, criticism, and other people's ideas. His honesty, his modesty, and his wicked sense of humor made it a pleasure to work with him. His wide interests outside of technology and his exceptional social skills made it easy to connect with him which resulted in many deep friendships reaching beyond the scope of work.

The Brazilian lyricist Paolo Coelho wrote:

> *Never. We never lose our loved ones. They accompany us; they don't disappear from our lives. We are merely in different rooms.*

The academic and Linux kernel communities will always be accompanied by Daniel and by the traces he left in his work and in our hearts.

About Linutronix

Linutronix GmbH is one of the leading service providers for all aspects of Linux in an industrial environment. Ranging from board support package, to the development environment and browser application, to consultation services for ongoing projects, we offer the full range of support from one single source.

However, all-in-one solutions are only a part of what we can offer. We also provide support for individual aspects of your projects.

We are committed to actively pariticipate in the development of the Linux kernel and other Free and Open Source Software (FOSS) projects. Our team has significantly contributed to Preempt-RT, to the Linux kernel in general and to other FOSS projects. We support our team members, who have maintainer roles in FOSS projects, so they can fulfil their important duties within paid hours.

Therefore, the Real-Time Linux Collaborative Project contracted us to advance the integration of Preempt-RT into the mainline Linux kernel.

Imprint

Legal disclosure:
Information in accordance with §5 TMG
(German Telecommunications Service Law):

Linutronix GmbH
Bahnhofstrasse 3
88690 Uhldingen-Muehlhofen / Germany

Managing directors:
Heinz Egger, Thomas Gleixner, Sean Fennelly,
Jeffrey Schneiderman, Tiffany Silva

Contact information:
Telephone: +49 7556 25 999 0
Fax: +49 7556 25 999 99
E-Mail: info@linutronix.de

Register entry:
Registration in the commercial register (Handelsregister).
Register court: District Court of Freiburg i.Br.
Register number: HRB 700 806

VAT identification number:
VAT identification number in accordance with §27a of the
German VAT Act (Umsatzsteuergesetz): DE252739476

Design and layout:
CHRISKNEIFEL · Kommunikation
Marktplatz 10, 87616 Marktoberdorf / Germany
Telephone: +49 8342 8956525
E-Mail: kontakt@chriskneifel.de
www.chriskneifel.de


LINUTRONIX
LINUX FOR INDUSTRY