Linux PREEMPT-RT vs. Commercial RTOSs: How Big is the Performance Gap?

Hasan Fayyad-Kazan and Luc Perneel, *Vrije Universiteit Brussel*

Martin Timmerman, *Royal Military Academy Brussels*

***Abstract*— Real-time operating systems (RTOSs) are getting more and more important for different uses in industry and become an integral part of commercial products today. Currently, there are many types of RTOSs, either open source or commercial ones with less or more features and characteristics. The aim of this research is to benchmark the real-time (RT) behaviour and performance of an open source RTOS (Linux PREEMPT-RT v3.6.6-rt17) and two commercial ones (QNX and Windows Embedded Compact 7), where all of them fall in the same RTOS category: they use virtual memory techniques to protect the kernel from user space, and protect the user space applications from each other. Flat memory RTOS's are not in this category. The benchmark is based on experimental measurements' metrics such as thread switch latency, interrupt latency, sustained interrupt frequency, mutex and semaphore acquisition and release durations, and finally the locking behaviour of mutex. These tests are executed on an x86 platform (ATOM processor) following a test framework and using non-invasive measurement equipment. The results show that the Linux PREEMPT-RT in its current version 3.6.6 is starting to be a competitor against the tested commercial RTOSs.**

I. Introduction

Real-time is an often misunderstood and/or a misapplied property of operating systems. It doesn't mean fast; Real-time deals with *guarantees*, not with *raw speed* [1]. In other words, the system must be deterministic to guarantee timing behaviour in the face of varying loads (from minimal to worst case) [2].

Real-Time Operating Systems (RTOS) are getting more and more important for different uses in the industry, and most commonly for embedded systems projects. Currently, there are a lot of available RTOSs in the market, both open-source and proprietary. With this huge number of available RTOSs, users of such systems always have the question in their mind: which RTOS is the best to use for my project. Obviously, this depends on the project requirements. However, in all cases, the performance and behaviour of a RTOS is an important decision element in choosing a RTOS.

Our research provides a benchmark framework testing the real-time behaviour and performance of RTOSs. The results of these tests may not only help designers making justified RTOS choices but also help them with a good application design.

The evaluated RTOSs in this paper are: an open source RTOS (Linux PREEMPT-RT v3.6.6-rt17) and two commercial ones (QNX and Windows Embedded Compact 7). For this evaluation, a testing suite of five performance tests and one behaviour test are used. The performance tests are: thread switch latency, interrupt latency, sustained interrupt frequency, and semaphore and mutex acquire-release timing in contention case, while the behaviour test checks the mutex locking behaviour.
The evaluation of these RTOSs is done an ATOM based hardware platform.

II. Evaluated RTOSs

A. Linux PREEMPT-RT v3.6.6-rt17

Although Linux is a popular and widely used OS, the standard Linux kernel fails to provide the timing guarantees required by critical real-time systems [3]. To circumvent this problem, academic research and industrial efforts have created several real-time Linux implementations [4]. The most adopted solutions are RTLinux/RTCore [5], RTAI [6], Xenomai [7] and the PREEMPT-RT [8] patch. Each one of these real-time enhanced kernels has its internal architecture, its strength and weaknesses [9]. All these approaches operate around the periphery of the kernel, except PREEMPT-RT patch which is mainlined in the current kernel and used by great actors such as WindRiver in their Linux4 [10] solution.

Linux Preempt-RT (LinuxPrt) [11, 12] is a Linux real-time patch maintained by Ingo Molnar and Thomas Gleixner [13]. This patch is the most successful Linux modification that transforms the Linux into a fully preemptible kernel without the help of microkernel (the architecture implemented in RTAI or RTLinux) [14]. It allows almost the whole kernel to be preempted, except for a few very small regions of code

DOI: 10.5176/2251-3043_3.1.244

("raw_spinlock critical regions"). This is done by replacing most kernel spinlocks with mutexes that support priority inheritance and are preemptive, as well as moving all interrupts to kernel threads [11, 15].

B. Windows Embedded Compact 7

Windows Embedded Compact 7 (formerly known as Windows Embedded CE 7.0) is the seventh major release of Windows Embedded CE operating system [16]. It is a real-time OS, separate from the Windows NT line, and is designed to target enterprise specific tools such as industrial controllers and consumer electronics devices such as digital cameras, GPS systems and also automotive infotainment systems [16]. Compared with other operating systems, Windows CE is widely acknowledged and used because of its powerful functions such as flexible definition, wide hardware support and embedded network support. [17]

C. QNX Neutrino 6.5

QNX Operation System (QNX OS) is one of multitask real-time operation systems, which is built on microkernel and protected address space mechanism. The strong points of QNX are the fact it is real-time, extremely stable, and reliable. QNX OS has been applied in fields such as industry automation, aviation, automobile, telecom, and its excellent performances and behaviour are approved in those fields. [18]

III. EXPERIMENTAL SETUP

In this paper, we evaluate Linux PREEMPT-RT v3.6.6-rt17, QNX Neutrino 6.5 and Windows CE7.
In order to have maximum control on the Linux 3.6.6 behaviour, we decided to build the kernel ourselves using the *buildroot* tool together with the real-time patch rt17. For evaluating this Linux kernel, we use the **µClibc** version 0.9.33 library between the testing framework and the kernel. The choice of this interfacing library is important because user applications (when using POSIX calls) can access the real-time features of the kernel only if the interfacing C library supports them; otherwise, direct system calls in kernel space are needed, making the application code not portable.
The kernel is configured to use a high frequency timer source as clock generation and all power management is disabled. The test application uses *mlockall()* to assure that all test programs are locked into memory. Further, the application is statically linked and started from a RAM disk (tmpfs) to avoid swapping out read-only code pages. Finally the real-time run away protection is disabled by setting the kernel configuration parameter `/proc/sys/kernel/sched_rt_runtime_us` to -1. All these precautions are required to assure real-time behaviour.

The kernel image of Windows CE7 is built using Visual studio 2008 and platform builder. Here also, the real-time application has to be built in a special way in order to avoid page faults. For that, the application is defined as a module in the platform.bib configuration file. Further, the critical section object [19] is used for testing mutex behaviour and performance (as it is optimized for in-process use only).

For QNX Neutrino, the QNX Momentics Integrated Development Environment (IDE) together with the Board Support Package (BSP) provided by QNX for this hardware is used to build the QNX kernel image.
Again, to avoid any page faults in QNX, a certain application can request IO privileges using the ThreadCtl() call. It is even possible to enforce direct physical memory allocation for all requested virtual memory (avoiding page faults completely after memory allocations) for all applications in the system by setting a kernel parameter. This kernel parameter is set for all our QNX tests.

We conduct our evaluation tests on an ATOM-based platform (Advantech SOM-6760) with the following characteristics:

- CPU: Intel Atom Z530 running at 1.6 GHz
- 32 Kbytes L1 instruction cache
- 24 Kbytes L1 write back data cache
- 512 Kbytes 8-way L2 cache
- 1 core with disabled hyper-threading support
- 512 MB DDR2 RAM
- VMETRO P-Drive PCI exerciser (PCI interrupt D, local bus interrupt level 10)
- VMETRO PBT-315 PCI analyser

IV. TESTING PROCEDURES AND RESULTS

A. Measuring process

A PCI device-simulator (VMETRO P-Drive) is inserted as a peer for the OS and as such, generating measurement samples by device access. It also generates interrupts while doing the interrupts tests, in an independent non-synchronized way with the platform under test (PUT). The test software generates start and stop events by writing a 32-bit word on the PCI bus towards the P-Drive. The traffic on the PCI bus is then captured using another PCI-Device (VMETRO bus-analyser) which in turn gives us the timing information on the event durations in the system. The major advantage of this system is that the interrupt latency (from hardware interrupt line being toggled to the interrupt handler) can easily be measured and taken into account.
Note that in the test's comparison figures below, the lower values mean better quality, and all the values in the charts are in µs.

B. Performance metrics

A quick online survey of RTOS metrics maintained by third party consultants, students, researchers, and official records (of distributor companies) reveals that the following three characteristics are used to evaluate a RTOS solution [12]:

- Memory footprint
- Latency
- Services performance

Among these measurements, footprint provides an estimated usage of memory by a RTOS on an embedded platform. The other two characteristics need to be measured in order to provide various types of RTOS overhead or runtime performance. Latency is reported in two different ways: interrupt and scheduling, and services performance is the minimum time taken by the RTOS interface to complete a system call [12].

In this paper, we test latency and services performance metrics. Before presenting the evaluation tests and the obtained results, we always perform two preliminary tests (the first 2 tests below) to assure the accuracy and precision of the tests.

1) Tracing overhead

This test calibrates the tracing system overhead which is fundamentally hardware related. The results presented in this paper are corrected from this constant bias.
Tracing precision depends on the PCI clock (33MHz), as this is the minimum time frame that can be detected. As a consequence, the results in this paper are correct to +/- 0.2 µseconds and are therefore rounded to 0.1 µseconds. The Y-Axis's in the charts are all in µseconds.

2) Clock tick processing duration

This test examines the clock tick processing duration in the kernel. The results of this test are extremely important as the clock interrupt might disturb all other performed measurements. This is due to the fact that on the PUT, being a commercial PC motherboard, the tick timer has the highest priority. Using a tickless kernel will not prevent this from happening (it will only lower the number of occurrences).

Figure 1 shows a comparison of the average and maximum clock processing durations of each RTOS:

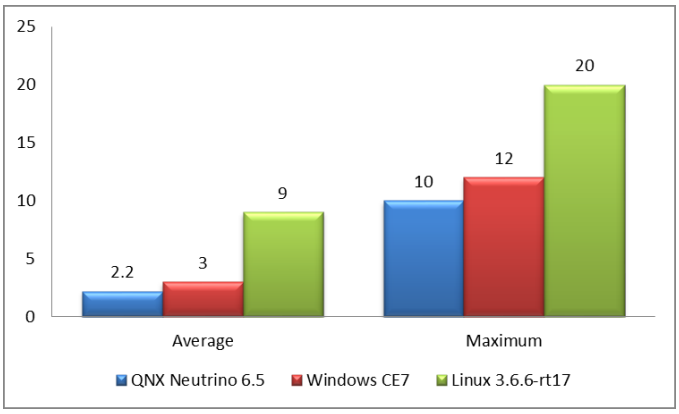

Figure 1: Average and maximum "clock tick durations" comparison

As we are focusing the testing on real-time behaviour and performance of these RTOSs, the maximum values out of the samples are our concern rather than the average ones. However, for the sake of completeness and comparison, we also publish the average values. The above chart results show that the commercial RTOSs (QNX and Windows CE) are much better than the Linux one. The maximum clock durations for QNX Neutrino 6.5 and Windows CE7 are in the same range(10 µs for QNX Neutrino and 12µs for Windows CE), while it is a higher value (20 µs) for the Linux RT.

Before going into the results' details, we give a brief explanation on how the tests are done and the raw measurements are corrected. Figure 2 is an example showing the results over time of the "switch latency between two threads" test performed on Linux 3.6.6. The X-Axis indicates the time when a measurement is performed with reference to the start of the test. The Y-Axis indicates the duration of the measured event.

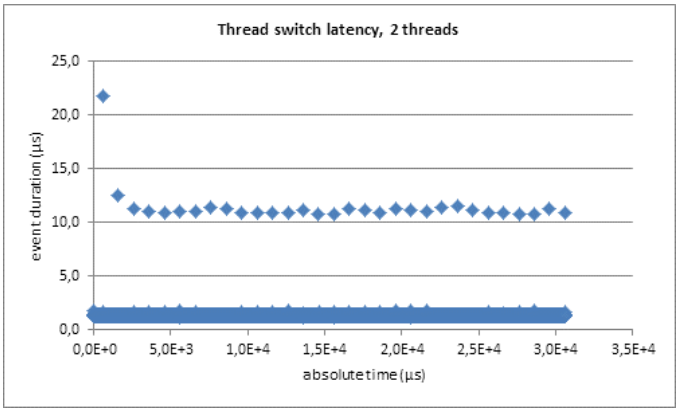

Figure 2: Thread switch latency between 2 threads, on Linux 3.6.6

Figure 2 shows that the average latency is 1.3 µs while the maximum latency is 21.8 µs, which is in this case at the beginning of the test. These values are actually the ones that are used in the comparison figures.

3) Thread switch latency between threads of same priority

This test measures the time needed to switch between threads of the same priority. For this test, threads must voluntarily yield the processor for other threads, so SCHED_FIFO scheduling policy is used. If we wouldn't use the "first in first out" policy, a round-robin clock event could occur between the yield and the trace, so that the thread activation is not seen in the trace. The test looks for worst-case behaviour and therefore it is done with an increasing number of threads, starting with 2 and going up to 1000. As we increase the number of active threads, the processor caching effect becomes visible (the 1000 threads scenario in figure 3a) as the thread context will no longer be able to reside in the cache with higher number of threads used (on this platform the L1 caches are 32KB and 24 KB, both for the instruction and the data caches respectively).Further, one will clearly see the influence of clock interrupts (figure 3b).

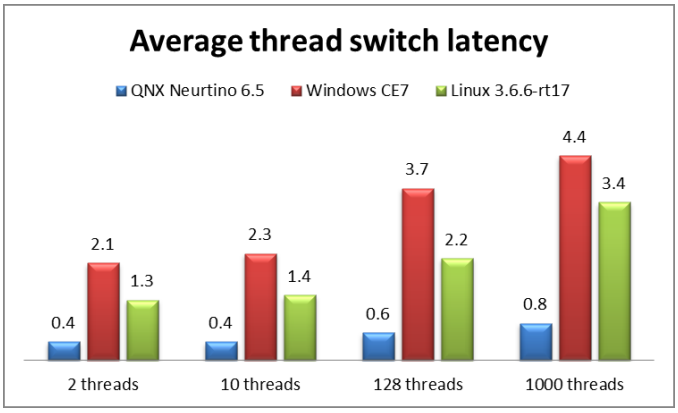

Figure 3a: Average "thread switch latency" comparison for different scenarios

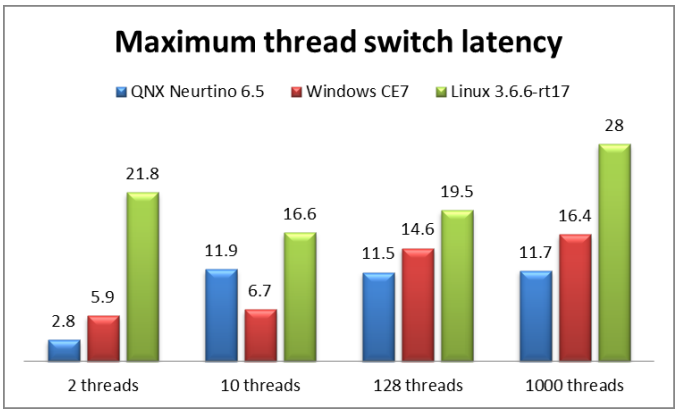

Figure 3b: Maximum "thread switch latency" comparison for different scenarios

Figure 3a shows that QNX Neutrino 6.5 outperforms both Linux and Windows for average latency, while Linux RT shows better results than Windows CE7.

The maximum thread latency (Figure 3b) depends also, in our configuration, on the clock tick interrupts which adds up to the bare maximum latency. A good RT hardware design should put the clock ticker on a minimum interrupt level to enhance these values. We observe that for Linux RT, the maximum latency is somewhat greater than QNX Neutrino and Windows CE7 in similar hardware circumstances.

4) Interrupt latency

This test measures the time it takes to switch from a running thread to an interrupt handler. Figure 4 shows a comparison between the average and maximum interrupt latencies for the RTOSs under test.

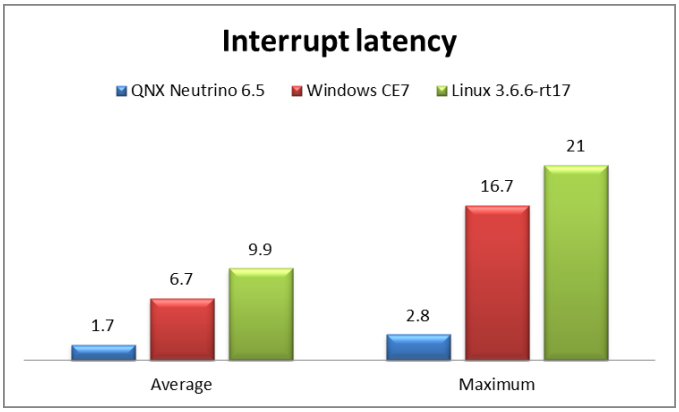

Figure 4: Average and maximum "interrupt latency" comparison

For real-time systems, the maximum interrupt latency (worst case duration) needs to be considered. Although one is never sure of obtaining the worst case duration via measurements, dramatically increasing the number of samples may help to get closer to the real worst case value. That's the reason why the long duration interrupt test (test metric 5) is catching a billion of interrupts. The following test is therefore considered of great importance and provides a simple RT-metric for comparison.

5) Maximum sustained interrupt frequency

This test detects when an interrupt cannot be handled anymore due to the interrupt overload. In other words, it shows a system limit depending on, for example, how long interrupts are masked, how long higher priority interrupts (the clock tick or other) take, and how well the interrupt handling is designed.

This test gives a very optimistic worst case value due to the fact that because of the high rate of interrupts, the amount of spare CPU cycles between the interrupts is limited or nil. Also, depending on the length of the interrupt handler, it might mostly be present in the caches. In a real world environment, the worst case will be greater.

However, the overhead of the cache misses in the interrupt handler compared with all other causes of delay will be rather small (at least in this case, where the worst case is much longer than the interrupt handler itself).

Unless the above considerations, this is a very popular test to compare RTOSs and see where the interrupt handling limit of this RTOSs is.

In this test, 1 billion interrupts are generated at specific intervals. Our test suite measures whether the system under test misses any of the generated interrupts. The test is repeated with smaller and smaller intervals until the system under test is no longer capable of handling this extreme interrupt load.

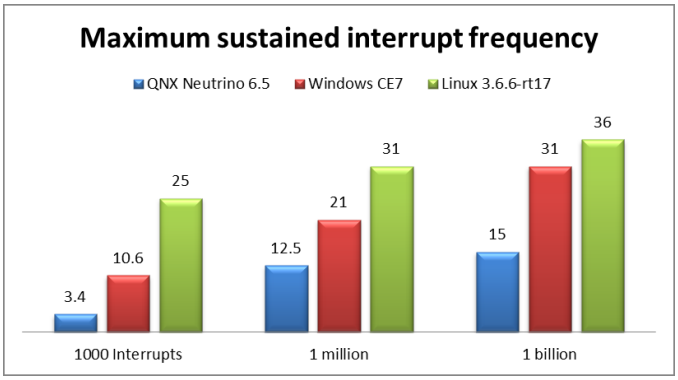

Figure 5: Maximum "sustained interrupt frequency" comparison

Figure 5 shows that if we only take a burst of 1000 interrupts, QNX Neutrino can handle all the generated interrupts without missing any until the duration between the generated interrupts is not less than 3.4 µs while it is 10.6 µs for Windows CE7 and 25 µs for Linux 3.6.6. Below these values, the tested RTOSs start to miss some interrupts.

Further on, the systems were tested by generating bursts of higher number of interrupts (1 million and 1 billion interrupts), which on the long run shows that the guaranteed interrupt duration for QNX is 15 µs (in case of 1 billion interrupts) while it is 31 µs for Windows CE7 and 36 µs for Linux 3.6.6. This shows that QNX is the best in handling the interrupts, while Windows CE7 and Linux 3.6.6 have a more than doubled value.

6) Semaphore acquire-release timings in the contention case

This test checks the time needed to acquire and release a semaphore, depending on the number of threads pending on the semaphore. In other words, it measures the time in the contention case when the acquisition and release system call causes a rescheduling to occur.

The purpose of this test is to see if the number of pending threads has an impact on the durations needed to acquire and release a semaphore. It attempts to answer the question: "How much time does the OS needs to find out which thread should be scheduled first and is this a constant time?"

In this test, as each thread has a different priority, the question is how the OS handles these pending thread priorities on a semaphore.

Here is the test scenario: 128 threads with different priorities are created (only 90 in Linux OS as it does not support 128 different real-time priorities). The creating thread has a lower priority than the created threads. When the created thread starts execution, it tries to acquire the semaphore; but as this semaphore is taken by the creating thread, the created thread blocks, and the kernel switches back to the creating thread. The time from the acquisition attempt (which fails) to the moment the creating thread is activated again is called here the "acquisition time". This time includes the thread switch time. Thread creation also takes time, which explains rather large values for the measurements.

After the last thread is created and pending on the semaphore, the creating thread starts to release the semaphore repeating this action the same number of times as the number of pending threads on the semaphore. The moment the semaphore is released, the "release duration" time is started. The highest priority thread that was pending on the semaphore will become active and it will stop the "release duration" time for the current pending thread. The "release duration" also includes the thread switch duration.

The results of this test show that the number of threads pending on a semaphore has NO impact on the release time periods, which is a good result for all evaluated RTOSs as this means that the RTOSs behave independently from the number of queued threads and as such keep a predictable response.

Figures 6a and 6b show a comparison between the average and maximum acquisition and release durations for the tested RTOSs.

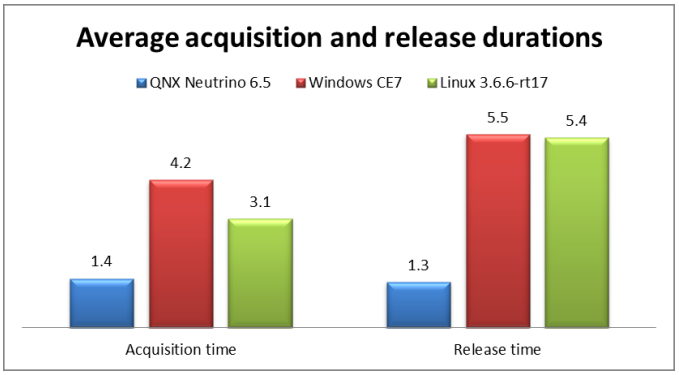

Figure 6a: Semaphore average "acquisition and release durations" comparison

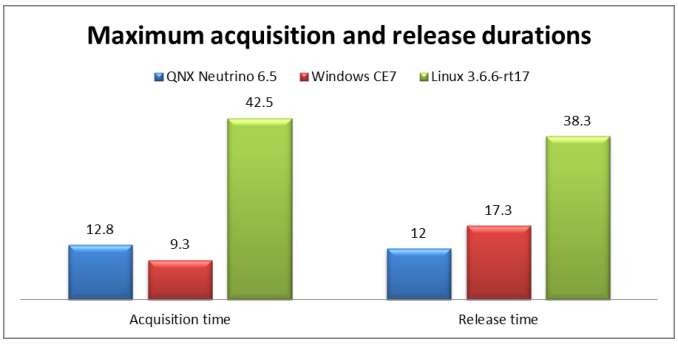

Figure 6b: Semaphore maximum "acquisition and release durations" comparison

Figure 6b shows that the maximum acquisition and release durations for Linux are bad. As we perform black box testing, it is hard to identify the cause of this issue, but it always happens around the clock tick. It looks like the clock tick handling duration in the operating system increases in such case. This is probably a potential enhancement for future Linux revisions.

7) Mutex Locking behaviour

This test checks the behaviour of the mutex locking primitive using the pthread_mutex_lock and related POSIX calls. Figure 7 helps us explaining this test which foresees both the priority inversion and priority inheritance cases.

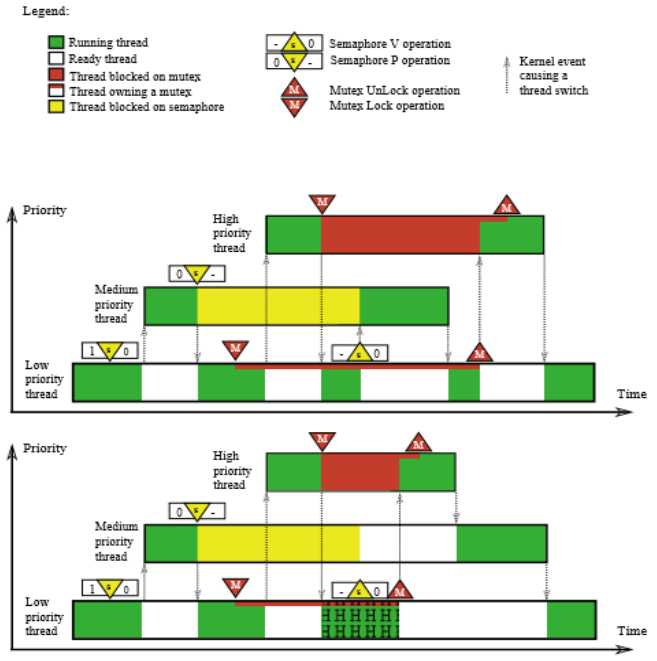

Figure 7: Priority inheritance and inversion cases

This test creates three threads. The low priority thread starts first. It creates a semaphore with count zero and starts the medium priority thread which activates immediately. The medium priority thread tries to acquire the semaphore, but as the semaphore count is zero, it blocks on the semaphore. The low priority thread continues execution; it creates a mutex for which it takes ownership and starts now a high priority thread which activates immediately. The high priority thread tries to acquire the mutex owned by the low priority thread and blocks also. The low priority thread resumes operation. Now the low priority thread releases first the semaphore and then the mutex. Remark that all threads of the described test are locked to the same processor avoiding parallel execution on a multi-processor system.

In case the mutex supports priority inheritance (lower part of figure 7), whenever the high priority thread requests the mutex, the low priority thread that has the ownership of the mutex will inherit the priority of the high priority thread that requested for the mutex ownership, and will do its job at this high priority. As a result, the low priority thread will first release the semaphore which will not wake-up the medium priority thread because the low priority thread is still running at high priority level and the low priority thread will continue its execution until it releases the mutex that was requested by the high priority thread. After releasing the mutex, the high priority thread will unblock and the low priority thread goes back to its normal priority level execution state (low priority level). In this case, the high priority thread becomes active, preempts the low priority thread and executes until it finishes its job. In the non-priority inheritance case (upper part of the figure), the medium priority thread will start upon the semaphore release and actually block the high priority thread for a long time, which results in a priority inversion.

We do not deal with the priority ceiling mechanism in this paper because the tested RTOSs limit their support to the priority inheritance mechanism.

8) Mutex acquire-release timings in the contention case

This test is comparable to the previous test metric 7, but performed in a loop. Here we measure the time needed to acquire and release the mutex in the priority inversion case. This test is designed in order to enforce a thread switch during the acquisition:

- The acquiring thread is blocked
- The thread with the lock is released.

The acquisition time starts from the moment a mutex acquisition is requested by a thread until the activation moment of the lower priority thread with the lock.

Note that before the release, an intermediate priority level thread is activated (between the low priority one owning the lock and the high priority one asking the locked resource). Due to the priority inheritance, this thread does not start running (the low priority thread owning the lock, inherited the high priority of the thread requesting the locked resource).

The release time is measured from the moment of the release call until the moment the thread requesting the mutex is activated. This measurement also includes a thread switch.

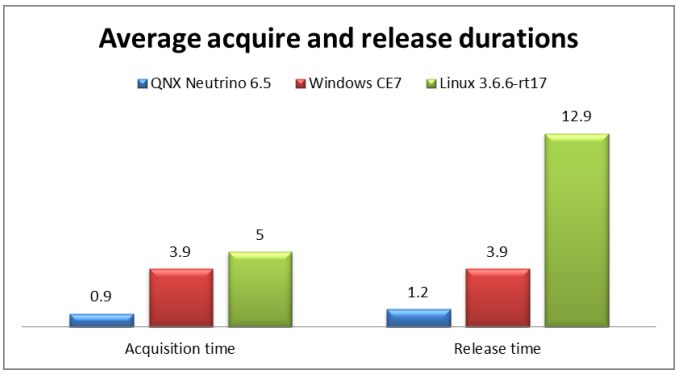

Figure 8a: Mutex average "acquire and release durations" comparison

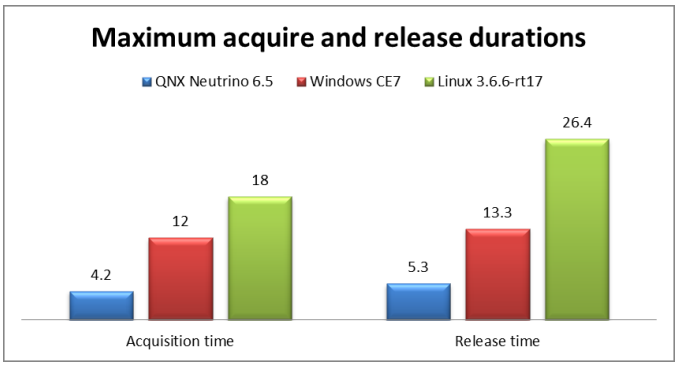

Figure 8b: Mutex maximum "acquire and release durations" comparison

The very short thread switch latency of QNX is remarkable here. The maximum latency is the sum of three components: mutex handling, thread switch latency and clock tick duration. The later one has its impact on the Linux latencies.

V. Conclusion

Analysing the obtained results show that the commercial RTOSs have still an advantage over the open source Linux RT_PREEMPT variant. Although this advantage is diminishing compared with results of a couple of years ago [20], the difference between Linux RT_PREEMPT and the best performing commercial RTOS of this evaluation study is still more than a factor of two.

Be aware that these results are only obtainable with a correctly configured Linux kernel and run-time environment, which needs serious Linux skills. Further, it has to be assured that none of the used drivers (kernel modules) behave badly if it comes to real-time behaviour. Overall, the tests show that a good behaving open source Linux RT_PREEMPT system is feasible and there is still room for enhancements in the future.

References

[1] K. Yaghmor, J. Masters, G. Ben-Yossef and P. Gerum, Building Embedded Linux Systems, 2nd Edition, Building Embedded Linux Systems, 2nd Edition, 2008.

[2] T. Jones, "Anatomy of real-time Linux architectures," IBM, [Online]. Available: http://www.ibm.com/developerworks/linux/library/l-real-time-linux/.

[3] P. Regnier, G. Lima and L. Barreto, "Evaluation of Interrupt Handling Timeliness in Real-Time Linux Operating Systems," ACM SIGOPS Operating Systems Review, vol. 42, no. 6, pp. 52-63, 2008.

[4] Z. Chen, X. Luo and Z. Zhang, "Research Reform on Embedded Linux's Hard Real-Time Capability in Application," in Proceedings of the 2008 International Conference on Embedded Software and Systems Symposia, 2008.

[5] " High Performance and Deterministic System Software-FSM Labs" [Online]. Available: http://www.fsmlabs.com/

[6] "RTAI-Official website," [Online]. Available: https://www.rtai.org/.

[7] "Xenomai," [Online]. Available: http://www.xenomai.org/

[8] M. Mossige, P. Sampath and R. Rao, "Evaluation of Linux rt-preempt for embedded industrial devices for Automation and Power Technologies - A Case Study," in Proceedings of the Ninth Real-Time Linux Workshop, Linz, 2007.

[9] N. Litayem and S. Ben Saoud, "Impact of the Linux Real-time Enhancements on the System Performances for Multi-core Intel Architectures," International Journal of Computer Applications, vol. 17, no. 3, 2011.

[10] W. River, "The First with the Latest: Wind River Linux 4," [Online]. Available: http://www.windriver.com/announces/linux4/. [Accessed 24 June 2012].

[11] P. McKenney, "A realtime preemption overview," 2005. [Online]. Available: http://lwn.net/Articles/146861/.

[12] S. Rostedt and D. V. Hart, "Internals of the RT Patch," in Proceedings of the Linux Symposium, 2007.

[13] "CONFIG PREEMPT RT Patch-RT wiki," [Online]. Available: https://rt.wiki.kernel.org/index.php/CONFIG_PREEMPT_RT_Patch.

[14] D. Kang, W. Lee and C. Park, "Kernel Thread Scheduling in Real-Time Linux for Wearable Computers," ETRI Journal, vol. 29, 2007.

[15] S. Arthur, C. Emde and N. Mc Guire, "Assessment of the Realtime Preemption Patches (RT-Preempt) and their impact on the general purpose performance of the system," in Real-Time Linux workshop, Linz, Austria, 2007.

[16] Wikipedia, "Windows Embedded Compact 7," [Online]. Available: http://en.wikipedia.org/wiki/Windows_Embedded_Compact_7. [Accessed 25 December 2012].

[17] Y. ZHANG, D. XUE, C. WU and P. JI, "Research of portable Information Terminal Based on MIPS processor and Windows CE," in *International Conference on Advanced Computer control*, 2008.

[18] S. XU, P. Haipeng , J. Ren and J. Su, "Design of the Modbus communication through serial port in QNX operation system," in

ISECS International Colloquium on Computing, Communication, Control, and Management, 2008.

[19] "Critical Section Objects" [Online]. Available: http://msdn.microsoft.com/en-us/library/ee482979%28v=winembedded.60%29.aspx

[20] Dedicated-Systems, [Online] Available: http://download.dedicated-systems.com/

Authors:

Hasan Fayyad-Kazan
Electronics and Informatics Department
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel-Belgium
{hafayyad@vub.ac.be}

Luc Perneel
Electronics and Informatics Department
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel-Belgium
{luc.perneel@vub.ac.be}

Martin Timmerman
Mathematics Department
Royal Military Academy Brussels
Hobbemastraat 8, 1000 Brussels-Belgium
{martin.timmerman@rma.ac.be}