Comparison of QNX Neutrino and Linux Vanilla operating systems on PowerPC processor

Copyright

Disclaimer

Authors

Luc Perneel (1, 2), Hasan Fayyad-Kazan(2) and Martin Timmerman (1, 2, 3)
1: Dedicated Systems Experts, 2: VUB-Brussels, 3: RMA-Brussels

http://download.dedicated-systems.com E-mail: info@dedicated-systems.com

EVALUATION REPORT LICENSE

Contents

1 About the RTOS evaluation project 4
1.1 Purpose and scope of the RTOS evaluation 4
1.2 Test framework used: 2.9 4
2 About the OSs and the testing platform 5
2.1 Software 5
2.2 Hardware 6
3 Evaluation results overview 7
3.1 Dedicated Systems' ratings for the tested (RT)OSs 7
3.2 Rating Criteria 8
3.3 Positive and negative points for each OS 9
Ratings by category 10
3.4.1 QNX Neutrino 6.5 with Patch 2530 11
3.4.3 Linux 2.6.32 with Board vendor patches 11
3.4 Tests Summary 12
3.5.1 Clock tick processing duration (CLK-P-DUR) 12
3.5.2 Thread switch latency between same priority threads (THR-P-SLS) 13
3.5.3 Maximum sustained interrupt frequency (IRQ_S_SUS) 14
3.5.4 Mutex acquire-release timings: contention case (MUT-P-ARC) 15
3.5.5 Mutex acquire-release timings: no-contention case (MUT-P-ARN) 16
4 Detailed Comparison 17
4.1 Clock tests (CLK) 17
4.2 Thread tests (THR) 18
4.2.1 Thread creation behaviour (THR-B-NEW) 18
4.2.2 Round robin behaviour (THR-B-RR) 19
4.2.3 Thread switch latency between same priority threads (THR-P-SLS)(TBD) 20
4.2.4 Thread creation and deletion time (THR-P-NEW) 22
4.2 Semaphore tests (SEM) 24
4.3.1 Semaphore locking test mechanism (SEM-B-LCK) 24
4.3.2 Semaphore releasing mechanism (SEM-B-REL) 25
4.3.3 Time needed to create and delete a semaphore (SEM-P-NEW) 25
4.3.4 Test acquire-release timings: non-contention case (SEM-P-ARN) 27
4.3.5 Test acquire-release timings: contention case (SEM-P-ARC) 28
4.3 Mutex tests (MUT) 30
4.4.1 Priority inversion avoidance mechanism (MUT-B-ARC) 30
4.4.2 Mutex acquire-release timings: no-contention case (MUT-P-ARN) 31
4.4.3 Mutex acquire-release timings: contention case (MUT-P-ARC) 33
4.4 Interrupt tests (IRQ) 35
5 Conclusion 36
6 Related documents 37

1 About the RTOS evaluation project

This section describes the purpose and scope of the evaluations conducted by Dedicated Systems.

1.1 *Purpose and scope of the RTOS evaluation*

This document provides quantitative measures to help potential RTOS users make objective comparisons between OSs and help them decide which OS is better for their needs.

This document compares the results of the quantitative evaluations of two (real time) operating systems (RTOSs):

- QNX Neutrino 6.5 patch 2530
- Vanilla Linux 2.6.32.13

The order in which we list these two OSs is based on the overall results obtained by the OSs, with the OS with the best results listed first. This ordering is maintained throughout the whole report.

Both (RT) OSs were evaluated on the same PowerPC platform (Freescale P1021 MDS).

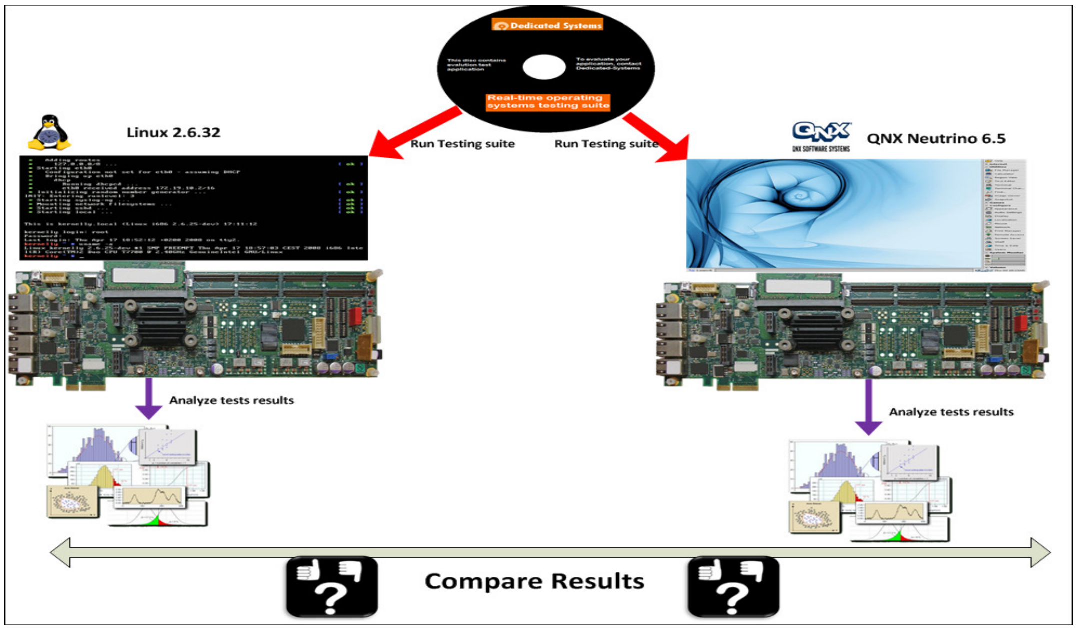


Figure 1: High level view of the evaluation procedure

1.2 *Test framework used: 2.9*

This document shows the test results in the scope of the evaluation framework 2.9. More details about this framework are found in Doc 1 (see section 6).

2 About the OSs and the testing platform

This section describes the OSs that Dedicated Systems tested using its Evaluation Testing Suite, and the hardware on which both OSs were running during the testing.

2.1 *Software*

The following table shows the operation systems' versions whose behavior and performance results were compared by Dedicated Systems after testing them with its evaluation testing suite on the same PowerPC platform (Freescale P1021MDS).

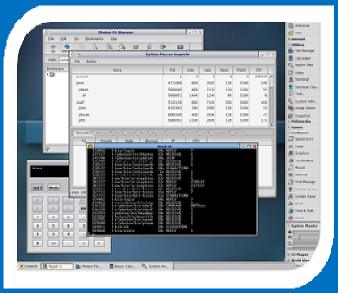
QNX Neutrino RTOS v6.5 with Patch 2530	Vanilla Linux 2.6.32
	

Table 1: The evaluated OSs

For **QNX Neutrino 6.5**, Patch 2530 was applied. This patch introduces a fix to the io-pkt network stack where a timer pulse implementation is used instead of attaching a handler to the timer interrupt. This patch significantly improves clock tick processing times and results in improved real time performance.

For **"Vanilla" Linux 2.6.32.13**, board vendor (Freescale) patches were applied.

2.2 Hardware

We conducted our tests on the same PowerPC platform. This platform is the Freescale QorIQ P1021 Modular Development System (MDS) board from Freescale with the following characteristics:

- Using the P1021 QorIQ™ communication processor.
- Power Architecture (P1021) dual core e500 processor running at 800 MHz (for the tests in this report, we disable one of the cores). As we use one core only, the results should be the same as on a P1012 board. The only difference between these two processors is the number of cores.
- L1 Cache: 32KB instruction and 32KB data cache (for each core)
- L2 Cache: 256KB (shared between cores, but tests run with one core only). Eight-way set-associative cache organization with 32-byte cache lines.
- 512MB DDR3 RAM (SODIMM) with ECC support running at 800MHz

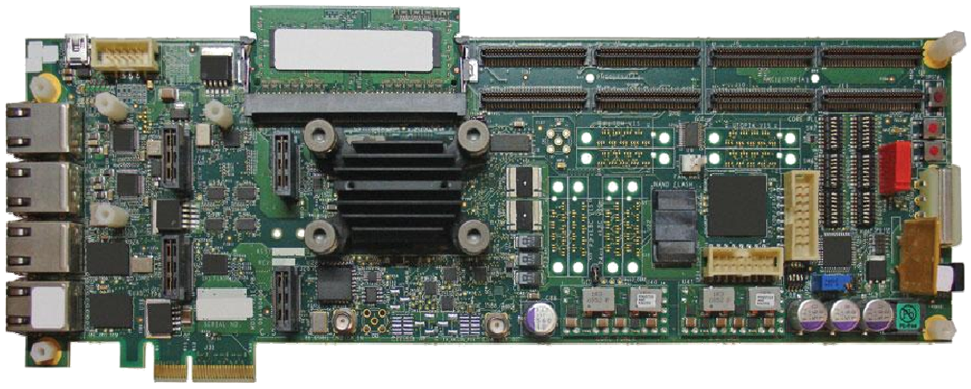


Figure 2: The P1021MDS board on which the tests were conducted

3 Evaluation results overview

This section presents the overall ratings and evaluations based on key tests.

3.1 *Dedicated Systems' ratings for the tested (RT)OSs*

Table 2 shows Dedicated Systems' overall ratings for the tested OSs:

QNX Neutrino 6.5	Vanilla Linux 2.6.32
★★★★★	★★½

Table 2: Overall ratings for the evaluated OSs

QNX Neutrino 6.5 test results show that QNX is an excellent real-time operating system, with good behavior in general and good real-time results in particular and with an efficient development environment around it.

For Linux, the test results need to be interpreted with care:

Knowing that we were unable in the allotted time budget to configure Linux completely (with all drivers) on this platform, we have been pushed to do the tests with a very minimal set of drivers. As a consequence, the load on the system during the tests was much less than what we normally do, resulting in good test results, at least for the tests we were able to perform. They are not really comparable to the QNX test results which were made with full system load (all drivers installed).

Also, we did not manage to make the interrupt test run correctly in the given time budget and therefore these tests results are not reported. This was due to systematic stability problems when one wanted to add supplementary drivers to the system or due to non-available drivers in the different Linux versions we tried out.

Our work demonstrated that it is very difficult to make Linux in whatever version run correctly on this PPC platform. It seems that PPC is not really endorsed by the Linux community in the same way the X86 family is endorsed and more recently also the ARM family. This is the reason why we give a rather low score.

3.2 Rating Criteria

After testing each OS using the Dedicated Systems Evaluation Testing Suite, we used a star system to give each OS a rating based on the performance and behavior results. The maximum number of stars that an OS can receive is five (5).

Rating	Availability of real-time requirements	Performance, behavior and interrupts testing results
★★★★★	The OS works correctly out-of-the-box. No fear about the kernel configuration	Excellent (Hard RT is met)
★★★★	The OS works correctly out-of-the-box. No fear about the kernel configuration	Very good (Hard RT is met)
★★★	Special attention and knowledge are required to correctly configure the kernel	Good (only soft RT is met)
★★	Special attention and deep knowledge are required to correctly configure the kernel	Bad (even soft RT problems)
★	Correctly configuring the kernel is problematic	Problematic
😕	NO RT capabilities	Fails (for hard and soft RT applications)

Table 3: Criteria used to evaluate OSs

3.3 Positive and negative points for each OS

Evaluated OS	Positive points 🙂🙂🙂	Negative points 🙁🙁🙁
QNX Neutrino 6.5	1) Excellent architecture for a robust and distributed system. 2) Very fast and predictable performance. 3) Large number of (BSPs) and drivers can be easily downloaded. 4) The availability of documentation and support is very high. 5) Efficient and user friendly (IDE)	1) Not all code is available in source code. Customers can apply for source access.
Vanilla Linux 2.6.32.13	1) No license fees. 2) Source code available. 3) Extensible	1) Not all real-time characteristics are present under Vanilla, which could lead to longer delays in heavy loaded systems. 2) GPL is not completely free… 3) Setting up a complete embedded target from scratch is a daunting task.

Table 4: Positive and negative points found in each evaluated OS

Ratings by category

The table below (Table 5) presents the “ratings by category” comparison for the evaluated OSs. For a detailed description of the rating criteria, see [Doc. 2].

For more details about each OS, please see the relevant theoretical evaluation reports: QNX Neutrino 6.5 in [Doc 4], and Linux 2.6 in [Doc 6].

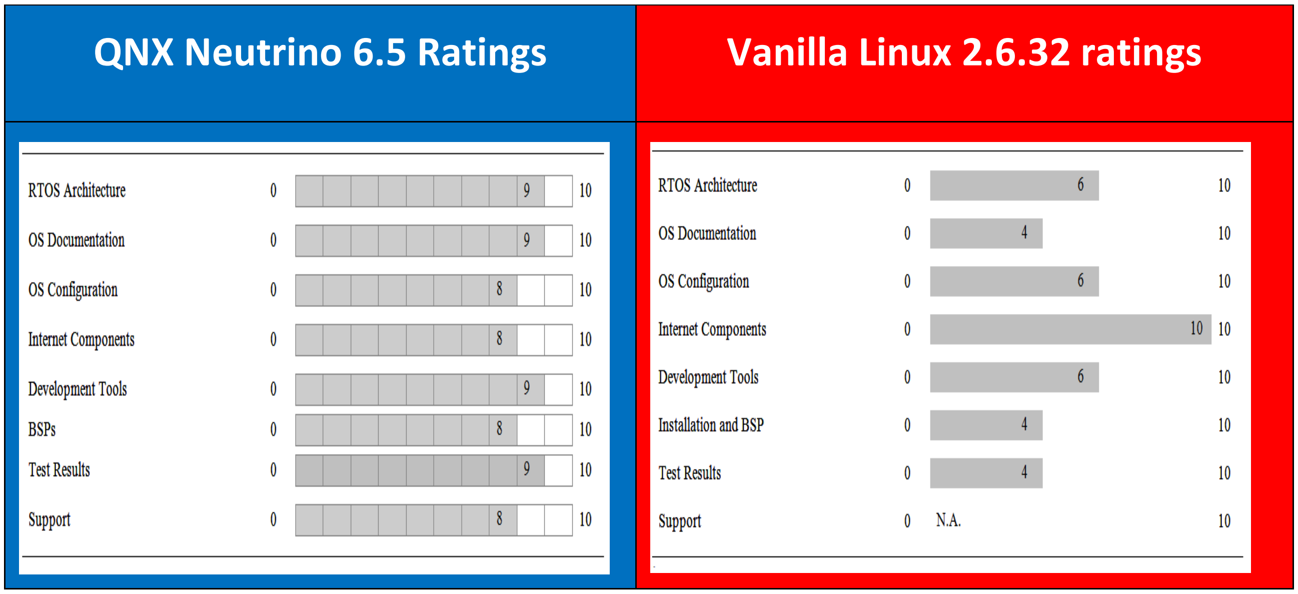


Table 5: Ratings of the evaluated OSs

3.4.1 QNX Neutrino 6.5 with Patch 2530

QNX Neutrino stands out as a clearly superior real-time OS compared to the other evaluated OS. In addition to its design, which is much more robust and very easy to debug, even at the driver level, the data from our tests for this OS confirm that its real-time behaviour is considerably better than that of the other OSs. Further, this OS it is very well documented, and users do not have to worry about kernel configuration, as the OS kernel is always configured correctly.

3.4.3 Linux 2.6.32 with Board vendor patches

The chief advantage of Linux is its open source licensing (no run-time fees). Note, however, that the GPL is not completely free, and investment is required to build a marketable system.
In this specific case, the hardware platform was not that well supported in Linux (the vendor should give back patches to the vanilla tree to improve this). Only the kernel delivered with the board from the vendor booted correctly. Different tries with custom made kernels/rootfs failed.
Also we had stability issues concerning the MMC devices. Further we did not succeed to get our interrupts tests working correctly (and we did not have enough free time to spend too much time on this issue).
So at the end this was not a good experience.

3.4 Tests Summary

This section presents a brief comparative summary of the most important evaluation tests performed on the OSs we tested.

Detailed comparisons can be found in the next chapter. More detailed information about each test and its importance can be found in the corresponding documents: QNX Neutrino 6.5 [Doc 5] and Linux 2.6.32 [Doc 7].

Note that in the comparison figures and tables:

- The lower values means better quality
- Values in the charts are in microseconds (μs)

3.5.1 Clock tick processing duration (CLK-P-DUR)

The "clock tick processing duration" test examines the clock tick processing duration in the kernel. The clock tick processing time is important because it impacts latencies everywhere in the system. The test results are extremely important because the clock interrupt will affect all the other measurements performed.

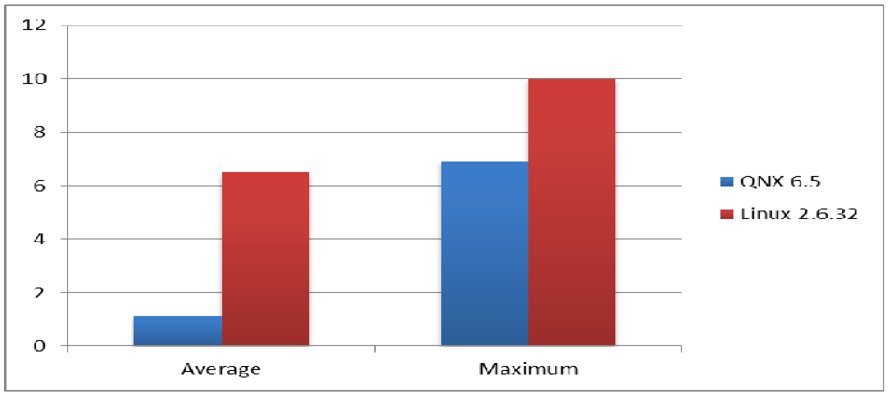


Figure 3: Average and maximum clock interrupt duration

Since we are interested in real-time behavior, the maximum values are more important than the average values. From our results, it is clear that QNX Neutrino 6.5 is still ahead of Vanilla Linux. The maximum clock duration for QNX Neutrino 6.5 is 6.9, while it is 10 for the Vanilla Linux.

3.5.2 Thread switch latency between same priority threads (THR-P-SLS)

The "latency between threads of same priority" test measures the time to switch between threads of the same priority using SCHED_FIFO policy. This test was performed four times, and each time using an increasing number of threads (2, 10, 128, and 1000) in order to generate the worst case behaviour.
The figures below present the thread switch latency with 1000 active threads in order to show the time values in the worst case. The evaluation results with fewer threads are presented in the next chapter.

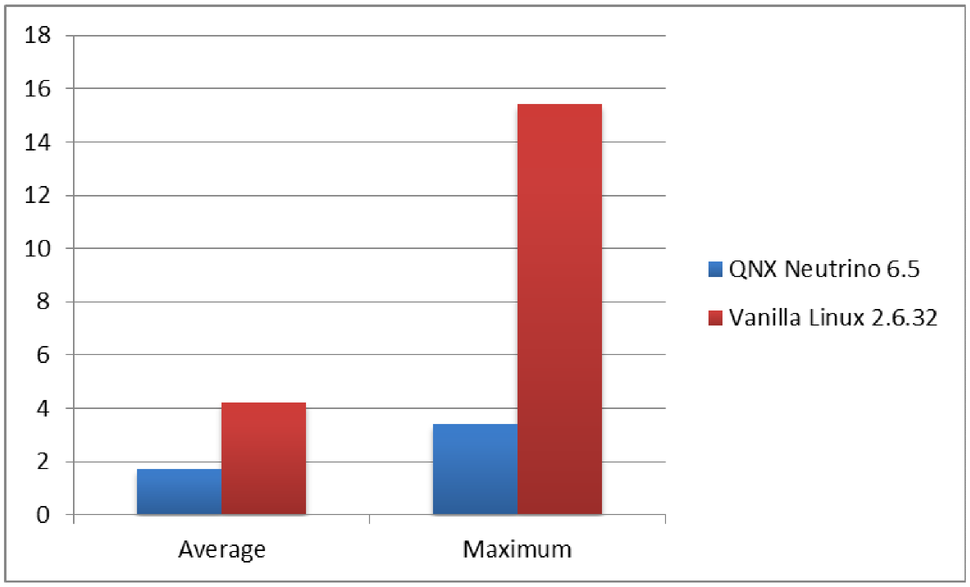


Figure 4: Average and maximum latency between 1000 threads

In this test, QNX Neutrino 6.5 outperforms Vanilla Linux 2.6.32 for average and maximum latency.

3.5.3 Maximum sustained interrupt frequency (IRQ_S_SUS)

The "maximum sustained interrupt frequency" test measures the probability that an interrupt might be missed. In other words, it attempts to answer the question: Is the interrupt handling duration stable and predictable?

In this test, 1 billion interrupts are generated at specific interval rates. 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 deemed to no longer handle the interrupt load.

For Vanilla Linux, we were not able to do the interrupts tests on this platform. The reason and the explanation for that can be found in the corresponding document (see Doc 7 in section 6 of this report) of evaluating Vanilla Linux on PowerPC on our website (download.dedicated-systems.com).

The interrupts tests for QNX Neutrino 6.5 on this platform can also be found on our website.

3.5.4 Mutex acquire-release timings: contention case (MUT-P-ARC)

The "mutex acquire-release timings in the contention case" test measures the time needed to acquire and release a mutex using priority inheritance. The acquire time is measured from the moment the higher priority thread requests the mutex until the moment the lower priority thread owning the mutex activates. The release time is measured from the moment the lower priority thread releases the mutex until the moment the higher priority thread is activated. As a result, the total time spent on a locked mutex is thus the sum of the acquisition time + release time + the time the lock is taken by the lower priority thread.

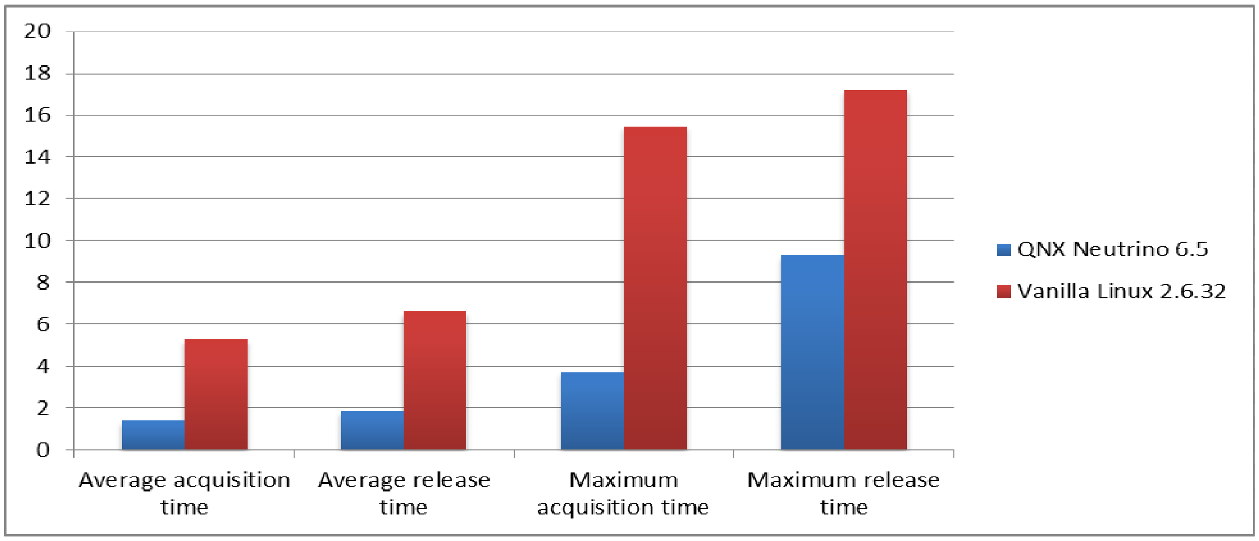


Figure 5: Mutex average and maximum acquire-release time: Contention case

The advantage of the classical RTOS (QNX Neutrino) compared with Linux is clear again.

3.5.5 *Mutex acquire-release timings: no-contention case (MUT-P-ARN)*

The "mutex acquire-release timings: no-contention case" test measures the overhead incurred using a lock when a thread is not locked by another thread.

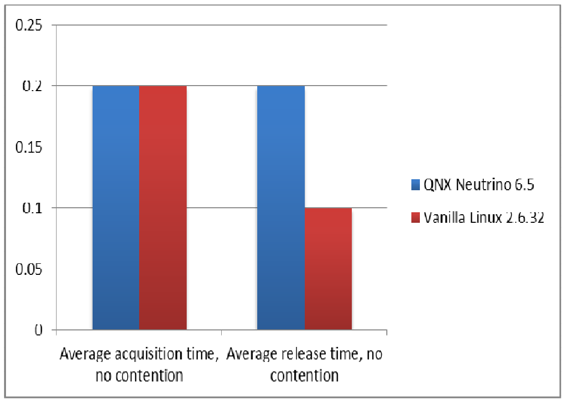


Figure 6a: Mutex average acquire-release time: no-contention case

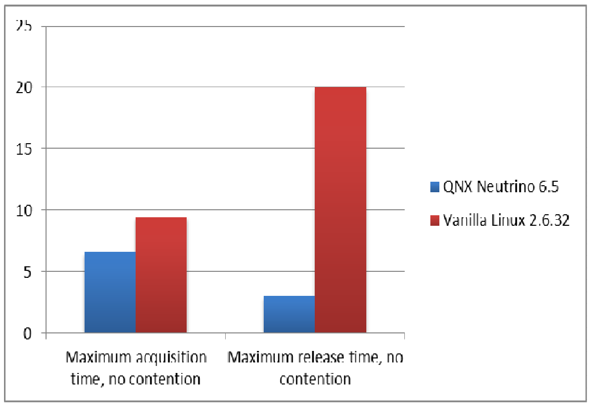


Figure 6b: Mutex maximum acquire-release time: no-contention case

In this case, the average values are important as they show how much time the locking overhead is in multi-threaded designs. Clearly, both OS tested here behave well. The maximum values are in this case not that important as they show us only the timer tick duration.

4 Detailed Comparison

This section presents the detailed test results and the comparison between the evaluated OSs.

4.1 *Clock tests (CLK)*

"Clock tests" measure the time that an operating system requires to handle its clock interrupts. On the tested platform, the clock tick interrupt is set on the highest hardware interrupt level, interrupting any other thread or interrupt handler.

4.1.1 Clock tick processing duration (CLK-P-DUR)

The "clock tick processing duration" test examines the clock tick processing duration in the OS kernel. The test results are extremely important, as the clock interrupt will affect all the other performed measurements. The table below shows the average and maximum clock interrupt duration for the tested OSs.

Clock interrupt duration	Average	Maximum
QNX Neutrino 6.5	1.1 µs	6.9 µs
Vanilla Linux 2.6.32	6.5 µs	10 µs

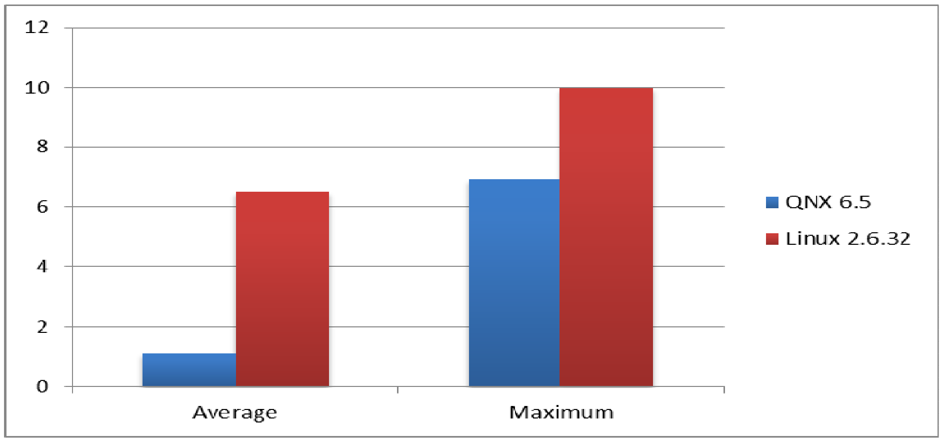


Figure 7: *Average* and *maximum* clock interrupt durations

The clock tick processing time is important because it impacts latencies everywhere in the system. As we are interested in real-time behavior, the measurements for maximum processing times are more important than the measurements for average processing times. Our testing showed that QNX Neutrino 6.5 performs better than Linux 2.6.32.

4.2 Thread tests (THR)

"Thread tests" measure the scheduler performance.

4.2.1 Thread creation behaviour (THR-B-NEW)

The "thread creation behavior" test examines the OS behavior when it creates threads. This test attempts to answer the question: Does the OS behave as it should in order to be considered a real-time operating system?

The following scenarios were checked in the test:

- **If a thread is created with a lower priority than the creating thread, can we be sure that it will not be activated until the creating thread is finished?**
- **If a thread is created with the same priority as the creating thread, is it placed at the end of the ready queue?**
- **When yielding after it was created by a thread of the same priority (as in the previous scenario), does the newly created thread becomes active?**
- **If a thread is created with a higher priority than the creating thread, does this new thread become activate immediately?**

QNX Neutrino 6.5	Vanilla Linux 2.6.32
Successfully passed this test	Successfully passed this test

Table 6: Results for the thread creation test

QNX Neutrino passed this test successfully without any problems.

However, in Linux we observed different behaviors depending on whether SCHED_FIFO or the SCHED_RR class was used. When lowering the priority of a thread, then this thread:

- is placed at the head of the ready queue if the Linux OS is running with SCHED_RR policy
- is placed at the end of the ready queue if the Linux OS is running with SCHED_FIFO policy

Note that changing priorities at run-time is equivalent to dynamically creating and deleting threads, something that should not be done in a real-time system.

4.2.2 Round robin behaviour (THR-B-RR)

The "round robin behavior" test checks if the scheduler uses a fair round robin mechanism to schedule threads that: use the SCHED_RR scheduling policy, are of the same priority, and are in the ready-to-run state (and using)!

QNX Neutrino 6.5	Vanilla Linux 2.6.32
Successfully passed this test	Passed ⚠

Table 7: Results of the round robin test

⚠ Note that:

- For the **Linux** scheduler, the initial time slice of a created thread is 10 times greater than other slices (1second instead of the default 100milliseconds (ms)).

Since dynamic thread creation and the use of different threads with the same priority is a poor practice in real-time systems, we assumed that real-time projects would avoid these practices. Given this assumption, we discounted these issues when we determined the final scores of the OSs we evaluated.

4.2.3 Thread switch latency between same priority threads (THR-P-SLS)(TBD)

The "thread switch latency between same priority threads" 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.

In this test, we use the SCHED_FIFO policy. If we do not 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.

This test was performed in order to generate the worst-case behavior. We performed the test with an increasing number of threads, starting with two (2) and going up to 1000 in order to observe the behavior in a worst-case scenario. As we increase the number of active threads, the caching effect becomes evident since the thread context will no longer be able to reside in the cache.

Test	QNX Neutrino 6.5		Vanilla Linux 2.6.32	
	Avg	Max	Avg	Max
Thread switch latency, 2 threads	0.7	6.1	1.4	21.4
Thread switch latency, 10 threads	0.7	8.1	1.7	28.3
Thread switch latency, 128 threads	1	4	3.4	14.2
Thread switch latency, 1000 threads	1.7	3.4	4.2	15.4

Table 8: Thread switch latency between x threads, in µs

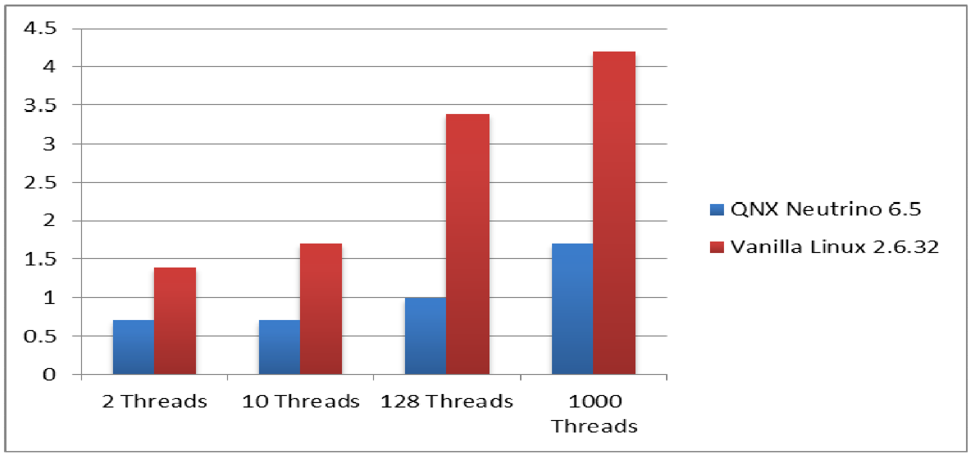


Figure 8a: Average switch latency between x threads, in µs

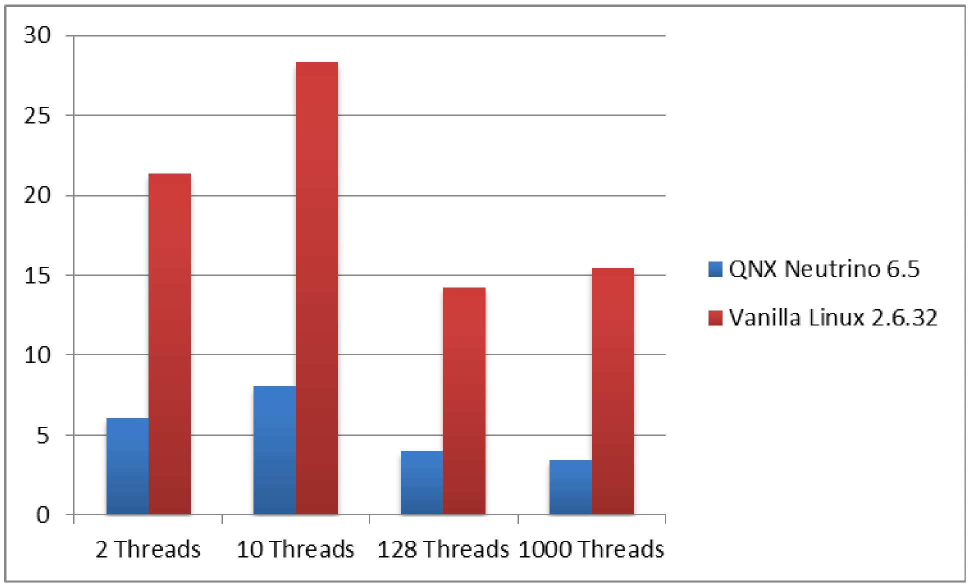


Figure 8b: Maximum switch latency between x threads, in µs

The impact of the caches on the average results is clearly observable (Figure 8a): the more threads there are to switch between, the more there are cache misses. The maximum values (Figure 8b) depend largely on the clock tick duration.

4.2.4 Thread creation and deletion time (THR-P-NEW)

The "thread creation and deletion time" test examines the time required to create a thread, and the time required to delete a thread in the following different scenarios:

- **Scenario 1 "never run": The created thread has a lower priority than the creating thread and is deleted before it has any chance to run. No thread switch occurs in this test.**
- **Scenario 2 "run and terminate": The created thread has a higher priority than the creating thread and will be activated. The created thread immediately terminates itself (thread does nothing).**
- **Scenario 3 "run and block": The same as the previous scenario (scenario 2: run and terminate), but the created thread does not terminate; it lowers its priority when it is activated, and therefore blocks.**

In the scenarios where the thread actually runs (2, 3), the creation time is the duration elapsed between the system call creating the thread and the moment the created thread is activated. For the "never run" scenario, the creation time is the duration of the system call.

Test		QNX Neutrino 6.5		Vanilla Linux 2.6.32	
		Avg	Max	Avg	Max
Never run	Thread creation	35	41.6	20.8	66.8
	Thread deletion	28	72.2	83.5	150.6
Run and terminate	Thread creation	34.2	38.5	41.3	96.1
	Thread deletion	3	14.2	1.2	51.2
Run and block	Thread creation	34.5	42	40.6	94.6
	Thread deletion	28.5	72.8	24.1	75.8

Table 9: Thread creation and deletion testing results, in µs

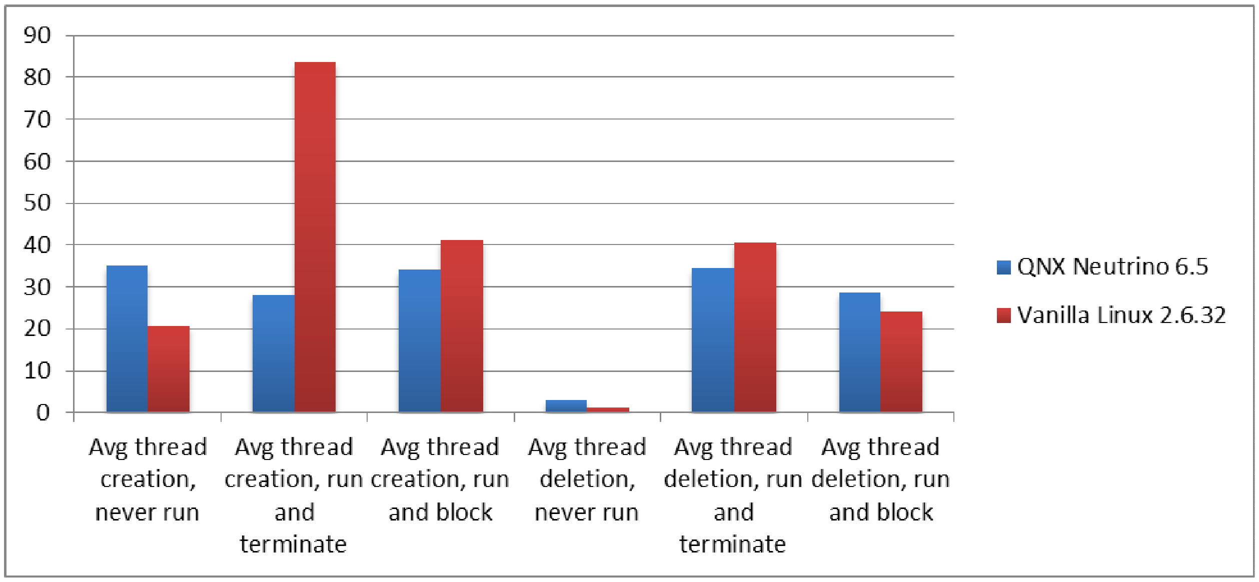


Figure 9a: Average thread creation and deletion times (µs) in different scenarios

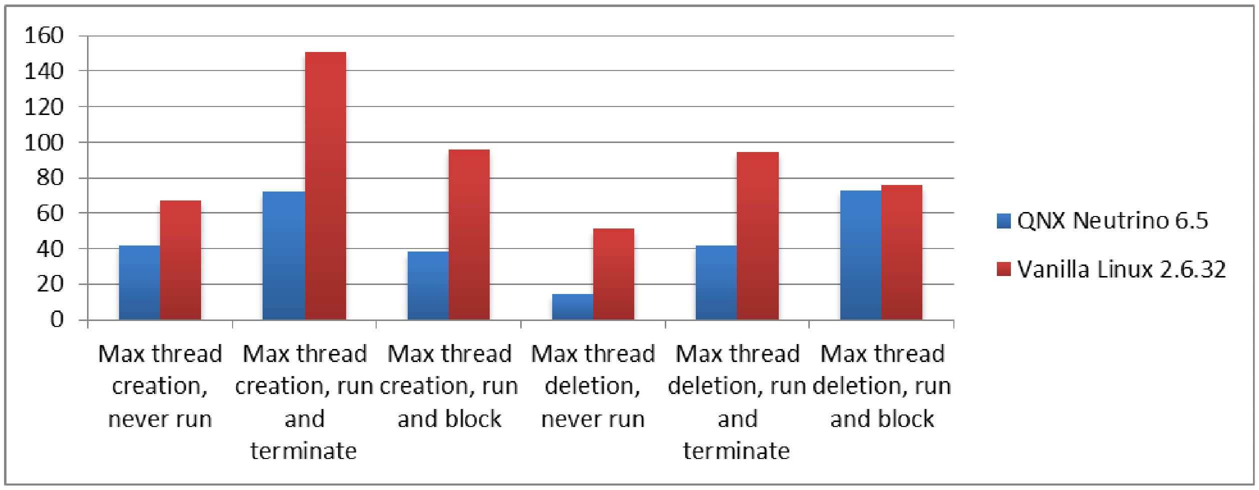


Figure 9b: Maximum thread creation and deletion times (µs) in different scenarios

As dynamic threads is a bad design principle in real-time systems, the measurements here are more informative and will not impact the real-time performance score.

4.2 Semaphore tests (SEM)

"Semaphore tests" examine the behavior and performance of the OS counting semaphore. The counting semaphore is a system object that can be used to synchronize threads.

With both tested operating systems, we did not specify a name to the semaphore when we conduct our tests. An unnamed semaphore cannot be used between processes. This limitation does not necessarily mean that the implementation with an unnamed semaphore does not use round-trips to the kernel.

4.3.1 Semaphore locking test mechanism (SEM-B-LCK)

In this test, we verify if the counting semaphore locking mechanism works as it is expected to work. If this mechanism works as expected, then:

- The `P ()` call will block only when the count is zero.
- The `V ()` call will increment the semaphore counter.
- In the case where the semaphore counter is zero, the `V ()` call will cause a rescheduling by the OS, and blocked threads may become active.

QNX Neutrino 6.5	Vanilla Linux 2.6.32
The semaphore behaves correctly as a protection mechanism	The semaphore behaves correctly as a protection mechanism

Table 10: Semaphore behaviour results

4.3.2 Semaphore releasing mechanism (SEM-B-REL)

The "semaphore releasing mechanism" test verifies that the highest priority thread being blocked on a semaphore will be released by the release operation. This action should be independent of the order of the acquisitions taking place.

QNX Neutrino 6.5	Vanilla Linux 2.6.32
Successfully passed this test	Successfully passed this test

Table 11: Semaphore release mechanism testing results

4.3.3 Time needed to create and delete a semaphore (SEM-P-NEW)

The "time needed to create and delete a semaphore" test measures the time needed to create a semaphore and the time needed to delete it. The deletion time is checked in two cases:

- **The semaphore is used between the creation and deletion.**
- **The semaphore is not used between the creation and deletion.**

Test		QNX Neutrino 6.5		Vanilla Linux 2.6.32	
		Avg	Max	Avg	Max
Semaphore is used	Creation time	1	15.2	0.1	8.8
	Deletion time	1	7.3	0.1	8.8
Semaphore is never used	Creation time	1	15.5	0.1	0.3
	Deletion time	1	7.7	0.1	9.2

Table 12: Semaphore creation and deletion time results

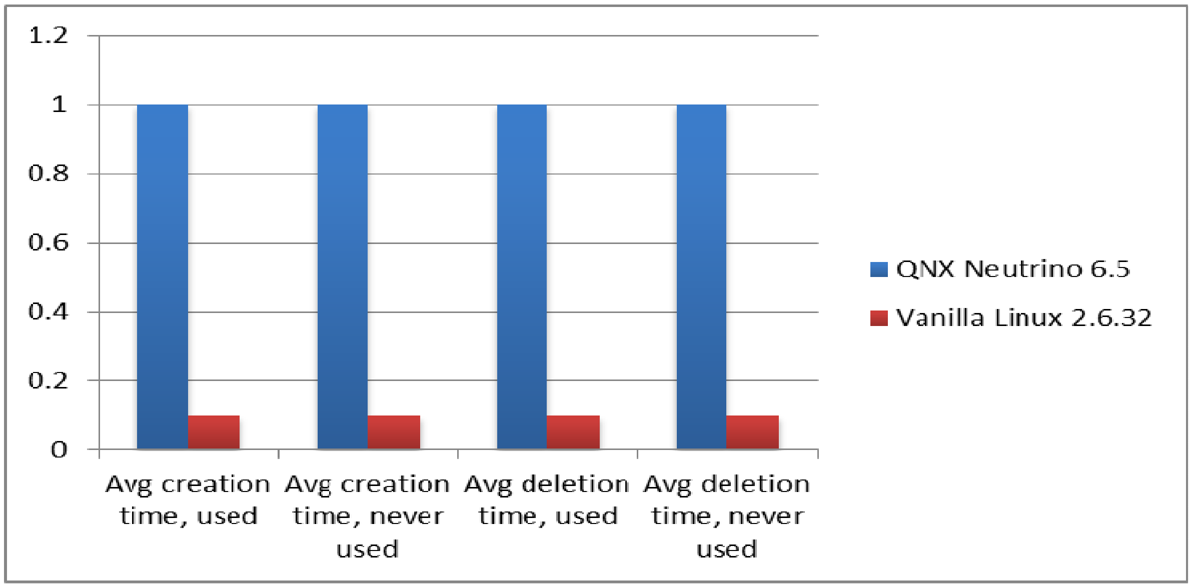


Figure 10a: Average creating and deleting times (µs) of a Semaphore in different cases

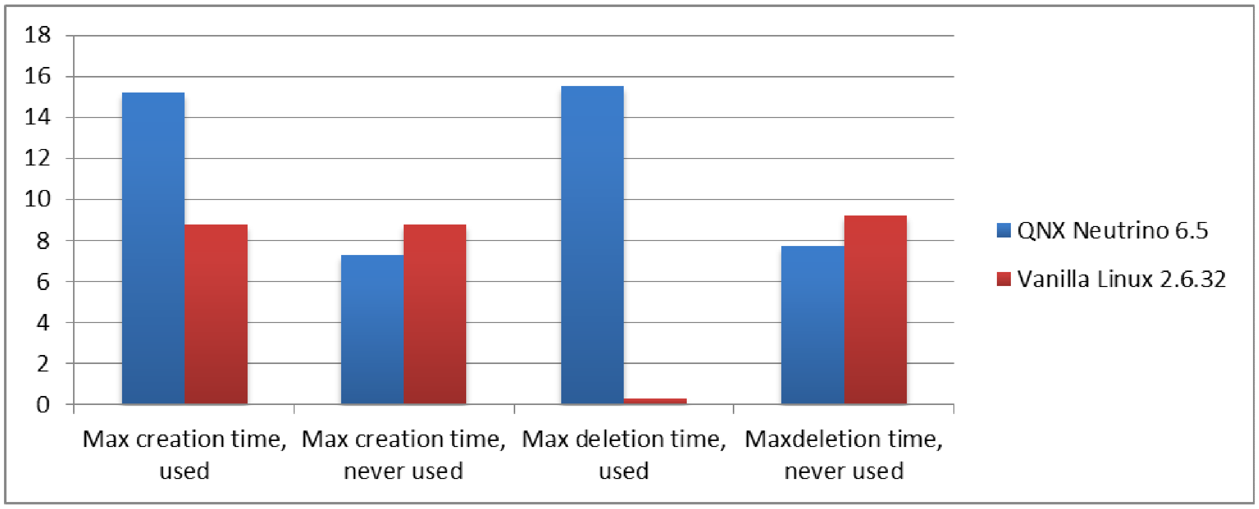


Figure 10b: Maximum creating and deleting times (µs) of a Semaphore in different cases

Remark that QNX Neutrino 6.5 uses inter-process (named) semaphores. Linux was tested with the traditional unnamed semaphores, which can only be used within a process. This causes the somewhat slower performance of QNX Neutrino here.

4.3.4 Test acquire-release timings: non-contention case (SEM-P-ARN)

The "acquire-release timings: non-contention case" test measures the acquisition and release time in the non-contention case. Since in this test the semaphore does not neither block nor causes any rescheduling (thread switching), the duration of the call should be short.

Test	QNX Neutrino 6.5		Vanilla Linux 2.6.32	
	Avg	Max	Avg	Max
Semaphore acquisition time, no contention	**0.7**	**6.2**	**0.1**	**8.9**
Semaphore release time, no contention	**0.7**	**4.3**	**0.1**	**8.9**

Table 13: Acquire release timings in the non-contention case

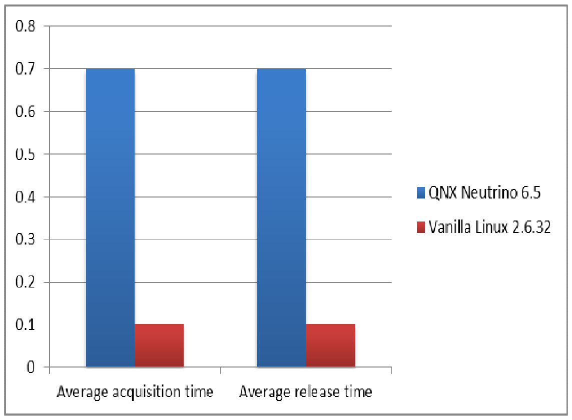


Figure 11a: Semaphore average acquire-release time: no contention

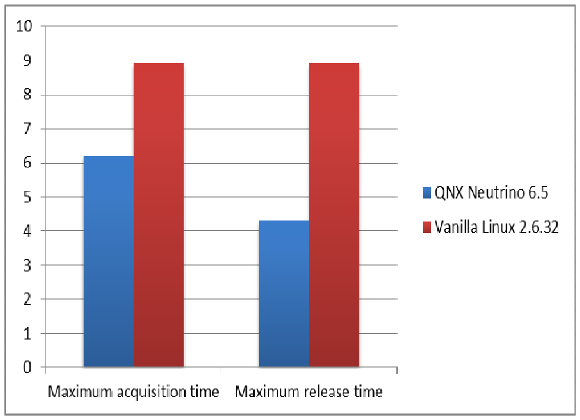


Figure 11b: Semaphore maximum acquire-release time: no contention

QNX performs a round trip to the kernel in these cases (like that required for named semaphores) while Linux uses atomic instructions and does not need a round-trip.

Note, however, that semaphores are much more appropriate way for signaling threads than for use as a protection mechanism. Indeed, semaphores do not have the concept of ownership and thus cannot be used to prevent priority inversions. If protection is required, mutexes should be used instead.

4.3.5 Test acquire-release timings: contention case (SEM-P-ARC)

The "acquire release timings: contention case" test is performed to test the time needed to acquire and release a semaphore, depending on the number of threads blocked on the semaphore. 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 blocked threads has an impact on the times 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?"

In this test, since each thread has a different priority, the question is how the OS handles these pending thread priorities on a semaphore. For more precise understanding of our test, please see the expanded diagrams showing a small time frame (e.g. one test loop). These diagrams are found in [Doc 5] for QNX Neutrino, and in [Doc 7] for Vanilla Linux. The test is conducted as follows:

- We create a semaphore with count zero: so it will block on acquire.
- We create 128 threads with different priorities. The creating thread has a lower priority than the threads being created.
- When a thread starts execution, it tries to acquire the semaphore; but as the semaphore is taken, the thread stops 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 takes some time; so the time between each measurement point is large compared with most other tests.
- After the last thread is created and is blocked on the semaphore, the creating thread starts to release the semaphore, repeating this action the same number of times as the number of blocked threads on the semaphore.
- We start timing from the moment the semaphore is released, which in turn activates the pending thread with the highest priority, which stops the timing. Again, the thread switch time is included in the measurement.

Test	QNX Neutrino 6.5		Vanilla Linux 2.6.32	
	Avg	Max	Avg	Max
Semaphore **acquisition** time, contention	**1.8**	**9.3**	**3.2**	**149.3**
Semaphore **release** time, contention	**2.3**	**14**	**7.8**	**136.7**

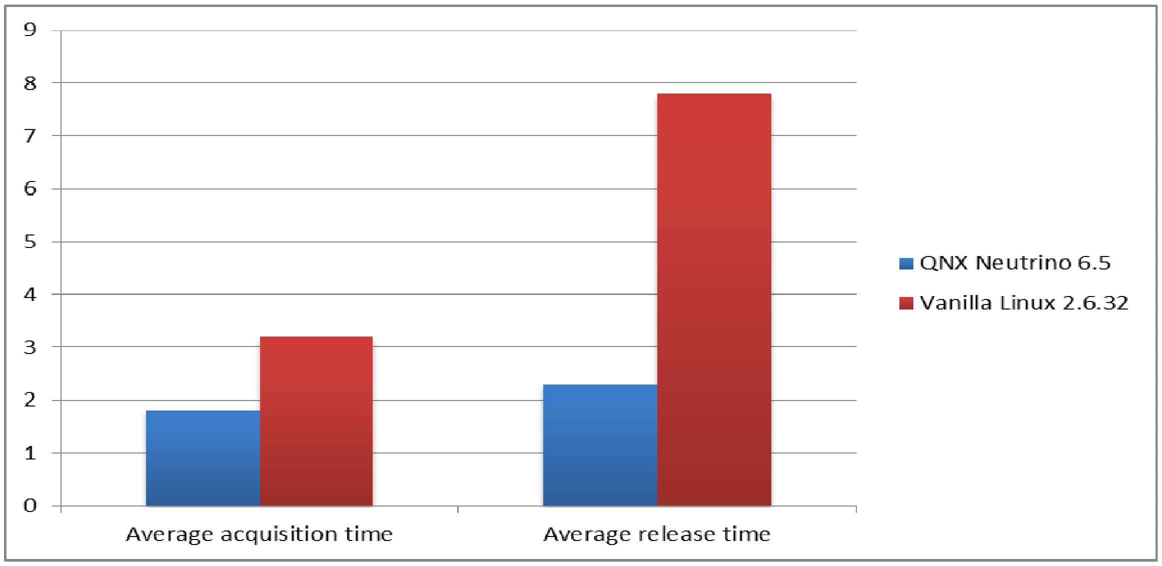


Figure 12a: Semaphore <u>average</u> acquire-release time: <u>Contention</u>

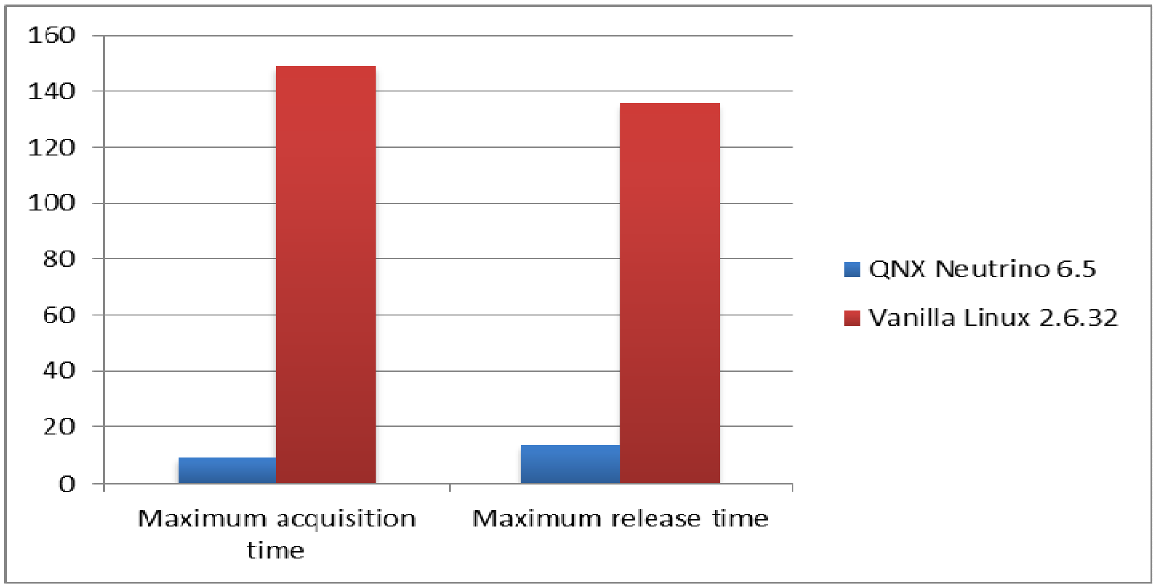


Figure 12b: Semaphore <u>maximum</u> acquire-release time: <u>Contention</u>

Linux RT exhibited a strange behaviour in this case. Clock tick interrupts seems to become much slower when acquiring/releasing a semaphore. So results are not good here for Vanilla Linux. The RT_PREEMPT patch does this much better.

4.3 Mutex tests (MUT)

Our "mutex tests" help us evaluate the behavior and performance of the mutual exclusive semaphore.

Although the mutual exclusive semaphore (further called mutex) is usually described as being the same as a counting semaphore where the count is one, this is not true. The behavior of a mutex is completely different than the behavior of a semaphore. Unlike semaphores, mutexes use the concept of a "lock owner", and can thus be used to prevent priority inversions. Semaphores cannot do this, and it goes without saying that mutexes (and not semaphores) should not be used semaphores for critical section protection mechanisms. In scope of the framework, this test will look into detail of a mutex system object that avoids priority inversion.

Our test will on purpose generate a priority inversion with three threads:

- Low priority thread having a lock
- Intermediate priority thread ready to run
- High priority thread running and requesting the lock owned by the low priority thread

If the mutex has some priority inversion avoidance mechanism present, the intermediate priority thread may not run until the lower priority thread released the mutex and the high priority thread finished its work.

Without such avoidance mechanism, the intermediate priority thread will start to run and thus delay the higher priority thread. Thus, as a result, priorities would be inverted!

4.4.1 Priority inversion avoidance mechanism (MUT-B-ARC)

The "priority inversion avoidance mechanism" test determines if the system call being tested prevents the priority inversion case. To check this possibility, the test artificially creates a priority inversion.

QNX Neutrino 6.5	Vanilla Linux 2.6.32
Priority inversion is prevented as expected	Priority inversion is prevented as expected

Table 14: results of priority inversion avoidance mechanism

4.4.2 Mutex acquire-release timings: no-contention case (MUT-P-ARN)

The "mutex acquire-release timings: no contention case" test measures the overhead incurred by using a lock when this lock is not owned by any other thread. Well-designed software will use non-contended locks most of the time, and only in some rare cases the lock will be taken by another thread.

Therefore, it is important that the non-contention case should be fast. Note that the required speed is only possible if the CPU supports some type of atomic instruction, so that no system call is needed when no contention is detected.

Test	QNX Neutrino 6.5		Vanilla Linux 2.6.32	
	Avg	Max	Avg	Max
Mutex **acquisition** time, no-contention	0.2	6.6	0.2	9.5
Mutex **release** time, no-contention	0.2	3	0.1	20

Table 15: Results of the mutex acquire-release timing in no-contention case, in µs

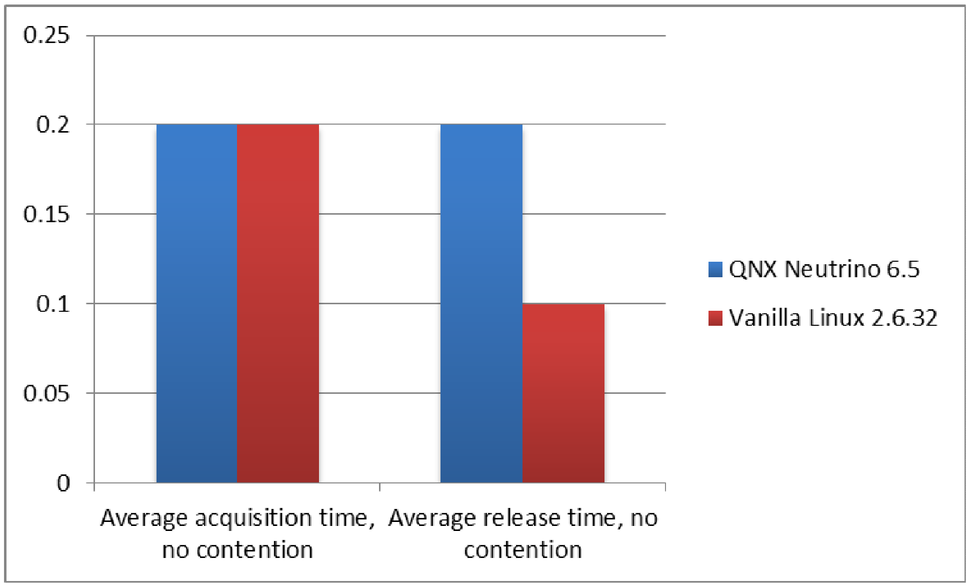


Figure 13a: Mutex average acquire-release time: no-contention

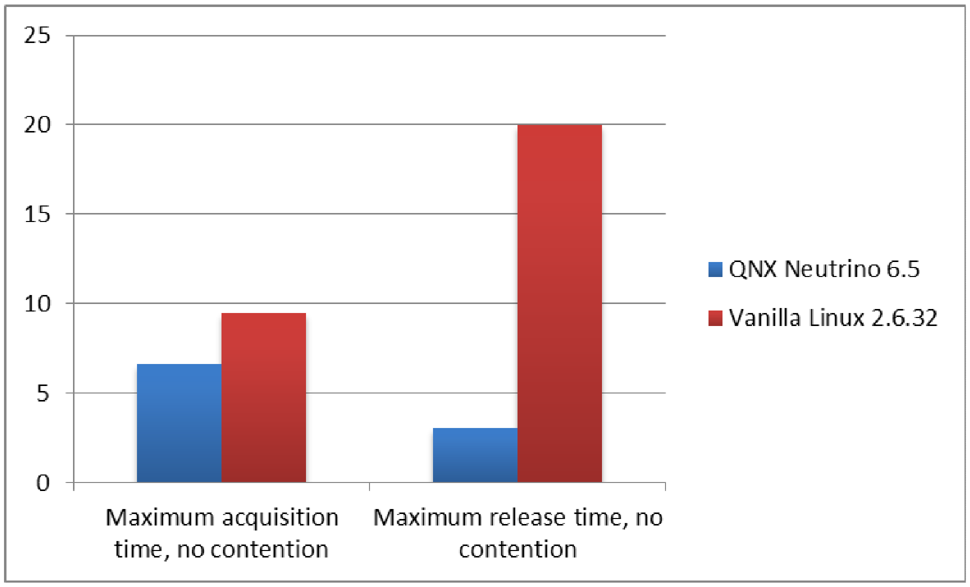


Figure 13b: Mutex maximum acquire-release time: no-contention

The average acquire-release time differences are too small to be measured. It means that both OSes tested here handle this well. The aim is indeed to have a minimal impact when using locks which, in a good design, will mostly be used without contention.

However, the maximum values were measureable: Figure 13b presents the clock tick durations, in microseconds (µs).

4.4.3 Mutex acquire-release timings: contention case (MUT-P-ARC)

The "mutex acquire-release timings: contention case" test is the same test as the "priority inversion avoidance mechanism (MUT_B_ARC)" test described above, but performed in a loop. In this case, we measure the time needed to acquire and release the mutex in the priority inversion case.

Our test is designed so that the acquisition enforces a thread switch:

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

We measured the acquisition time from the request for the mutex acquisition to the activation 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 having the lock and the high priority one asking the lock). Due to the priority inheritance, this thread does not start to run (the low priority thread having the lock inherited the high priority of the thread asking the lock).

We measured the release time from the release call to the moment the thread requesting the mutex was activated; so this measurement also includes a thread switch.

Test	QNX Neutrino 6.5		Vanilla Linux 2.6.32	
	Avg	Max	Avg	Max
Mutex **acquisition** time, contention	**1.4**	**3.7**	**5.3**	**15.5**
Mutex **release** time, contention	**1.9**	**9.3**	**6.6**	**17.2**

Table 16: Results of the mutex acquire-release timing in contention case, in µs

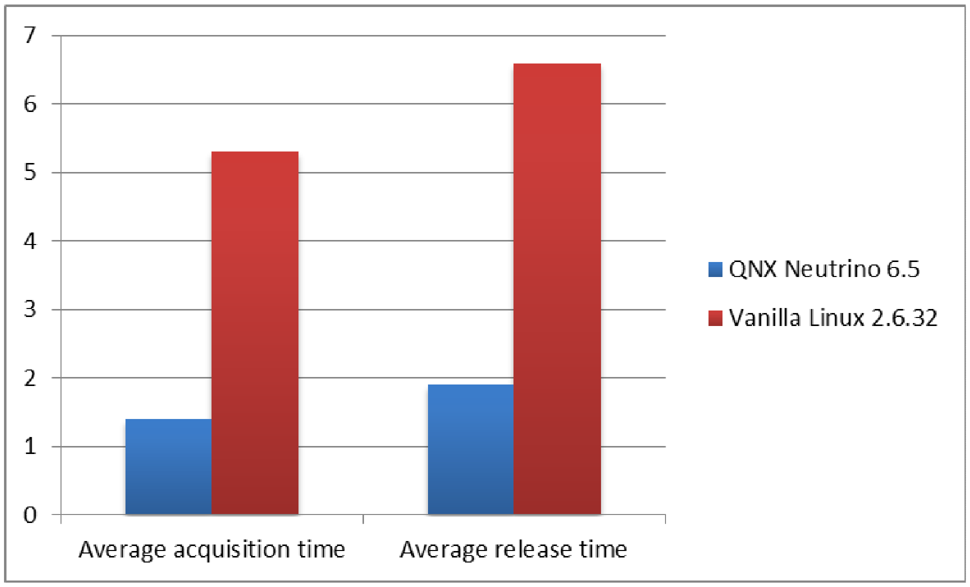


Figure 14a: Mutex average acquire-release time: contention

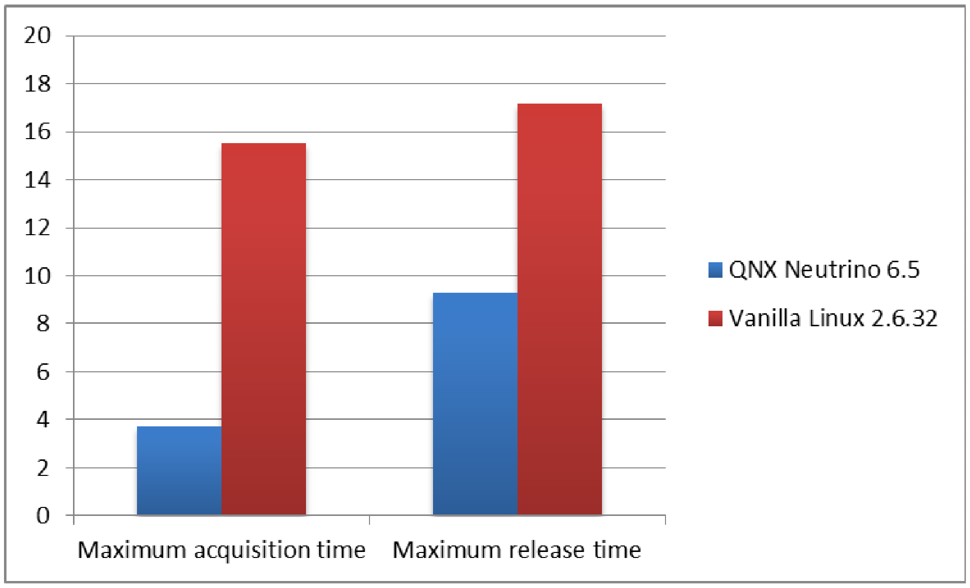


Figure 14b: Mutex maximum acquire-release time: contention

4.4 Interrupt tests (IRQ)

"Interrupt tests" evaluate how the operating system performs when handling interrupts.

For Vanilla Linux, we were not successful in doing the interrupts tests on this platform within the allotted timeframe. The reason and the explanation for this can be found in the corresponding document (see Doc 7 in section 6 of this report) of evaluating Vanilla Linux on PowerPC on our website (download.dedicated-systems.com).

The interrupts tests for QNX Neutrino 6.5 on this platform can also be found on our website.

5 Conclusion

QNX Neutrino 6.5 showed that it has an excellent architecture which made it a robust and trusted OS for meeting the resource requirements of real-time embedded systems. It is an out-of-the-box real-time operating system where everything is configured in correct way, so you do not have to worry about any settings and kernel configurations to meet the real-time requirements.

Employing our custom built Linux version on this platform was very difficult. We succeeded in employing a Linux version 3.x, but unfortunately most of the devices on this platform did not function correctly due to the non-availability of drivers for such platform. We managed to boot this platform with the kernel version supplied by the board provider, but this version was without real-time capabilities and we observed stability issues and only a limited number of devices were working and as such causing fewer disturbances in the measurements as one might expect.

All these prevented us from doing our tests in an easy way as we normally do with other Linux versions on x86 and ARM platforms, and after some weeks of inspecting and diminishing the difficulties, we succeeded in doing our evaluation tests with a subset system compared to what we normally do. The interrupt tests were also not done.

Our work demonstrated that it is very difficult to make Linux in whatever version run correctly on this PPC platform. It seems that PPC is not really endorsed by the Linux community in the same way the X86 family is endorsed and more recently also the ARM family. This is the reason why we gave a rather low score.

The ease of implementing and using Linux on a PPC platform depends on the platform vendor. For instance, do they make all driver/platform enhancements in the vanilla kernel tree available or not?

PPC Linux uses today a device tree (DTS) to configure hardware/software, which deviates from the classic approach on other architectures.

This impacts seriously how to write kernel modules on a PPC platform. Although the aim seems to be to simplify things, it makes it more difficult to make/reuse custom kernel modules on such platforms.

6 Related documents

These are documents that are closely related to this document. They can all be downloaded using following link: http://download.dedicated-systems.com/

Doc. 1	The evaluation framework This document presents the evaluation framework. It also indicates which documents are available, and how their name giving, numbering and versioning are related. This document is the base document of the evaluation framework. EVA-2.9-GEN-01	Issue: 1	Date: April 19, 2004
Doc. 2	The evaluation test report definition. This document presents the different tests issued in this report together with the flowcharts and the generic pseudo code for each test. Test labels are all defined in this document. EVA-2.9-GEN-03	Issue: 1	April 19, 2004
Doc. 3	The OS evaluation template This document presents the layout used for all reports in a certain framework. EVA-2.9-GEN-04	Issue: 1	April 19, 2004
Doc. 4	QNX v6.5, Theoretical evaluation report This document presents the qualitative discussion of the QNX OS EVA-2.9-OS-QNX-65	Issue: 4.1	Sept 8, 2011
Doc.5	QNX technical evaluation report This document presents the results of evaluating QNX on PPC platform EVA-2 9-TST-QNX-65-PPC-01	Issue: 1.07	Nov 7, 2011
Doc. 6	Linux theoretical evaluation report This document presents the qualitative discussion of the Linux OS EVA-2.9-OS-LINUXRT_2.6.33.7.2-rt30	Issue: 1.07	May 30, 2011
Doc.7	Linux technical evaluation report This document presents the results of evaluating Linux on PPC platform EVA-2_9-TST-LINUXRT_2_6_32-PPC	Issue: 1	May 2, 2012

Appendix A: Acronyms

Acronym	Explanation
API	Application Programmers Interface: calls used to call code from a library or system.
BSP	Board Support Package: all code and device drivers to get the OS running on a certain board
DSP	Digital Signal Processor
FIFO	First In First Out: a queuing rule
GPOS	General Purpose Operating System
GUI	Graphical User Interface
IDE	Integrated Development Environment (GUI tool used to develop and debug applications)
IRQ	Interrupt Request
ISR	Interrupt Servicing Routine
MMU	Memory Management Unit
OS	Operating System
PCI	Peripheral Component Interconnect: bus to connect devices, used in all PCs!
PIC	Programmable Interrupt Controller
PMC	PCI Mezzanine Card
PrPMC	Processor PMC: a PMC with the processor
RTOS	Real-Time Operating System
SDK	Software Development Kit
SoC	System on a Chip

DOCUMENT CHANGE LOG

Issue No.	Revised Issue Date	Para's / Pages Affected	Reason for Change
0	May 1, 2012	ALL	Initial report
1	May 10, 2012	ALL	Final Report