Behavior and performance evaluation of Linux 2.6.33.7.2-RT30 on ATOM

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

This is a legal agreement between you (the downloader of this document) and/or your company and the company DEDICATED SYSTEMS EXPERTS NV, Diepenbeemd 5, B-1650 Beersel, Belgium.
It is not possible to download this document without registering and accepting this agreement on-line.

1. **GRANT**. Subject to the provisions contained herein, Dedicated Systems Experts hereby grants you a non-exclusive license to use its accompanying proprietary evaluation report for projects where you or your company are involved as major contractor or subcontractor. You are not entitled to support or telephone assistance in connection with this license.
2. **PRODUCT**. Dedicated Systems Experts shall furnish the evaluation report to you electronically via Internet. This license does not grant you any right to any enhancement or update to the document.
3. **TITLE**. Title, ownership rights, and intellectual property rights in and to the document shall remain in Dedicated Systems Experts and/or its suppliers or evaluated product manufacturers. The copyright laws of Belgium and all international copyright treaties protect the documents.
4. **CONTENT**. Title, ownership rights, and an intellectual property right in and to the content accessed through the document is the property of the applicable content owner and may be protected by applicable copyright or other law. This License gives you no rights to such content.
5. **YOU CANNOT**:
   - You cannot, make (or allow anyone else make) copies, whether digital, printed, photographic or others, except for backup reasons. The number of copies should be limited to 2. The copies should be exact replicates of the original (in paper or electronic format) with all copyright notices and logos.
   - You cannot, place (or allow anyone else place) the evaluation report on an electronic board or other form of on line service without authorization.
6. **INDEMNIFICATION**. You agree to indemnify and hold harmless Dedicated Systems Experts against any damages or liability of any kind arising from any use of this product other than the permitted uses specified in this agreement.
7. **DISCLAIMER OF WARRANTY**. All documents published by Dedicated Systems Experts on the World Wide Web Server or by any other means are provided "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. This disclaimer of warranty constitutes an essential part of the agreement.
8. **LIMITATION OF LIABILITY**. Neither Dedicated Systems Experts nor any of its directors, employees, partners or agents shall, under any circumstances, be liable to any person for any special, incidental, indirect or consequential damages, including, without limitation, damages resulting from use of OR RELIANCE ON the INFORMATION presented, loss of profits or revenues or costs of replacement goods, even if informed in advance of the possibility of such damages.
9. **ACCURACY OF INFORMATION**. Every effort has been made to ensure the accuracy of the information presented herein. However Dedicated Systems Experts assumes no responsibility for the accuracy of the information. Product information is subject to change without notice. Changes, if any, will be incorporated in new editions of these publications. Dedicated Systems Experts may make improvements and/or changes in the products and/or the programs described in these publications at any time without notice. Mention of non-Dedicated Systems Experts products or services is for information purposes only and constitutes neither an endorsement nor a recommendation.
10. **JURISDICTION**. In case of any problems, the court of BRUSSELS-BELGIUM will have exclusive jurisdiction.

Agreed by downloading the document via the internet.

DOCUMENT CHANGE LOG

Issue No.	Revised Issue Date	Para's / Pages Affected	Reason for Change
0.1	14 April 2012	All	Initial version
1.0	16 April 2012	All	Modify some text
2.0	26 April 2012	All	Final

1 Document Intention

1.1 Purpose and scope

This document presents the quantitative evaluation results of the real-time **Linux** operating system (**Linux** with its real-time patches) on an ATOM-based platform. The testing results of this operating system employed on an ATOM processor can be found on our website. (www.dedicated-systems.com)

The layout of this report follows the one depicted in "The OS evaluation template" [Doc. 4]. The test specifications can be found in "The evaluation test report definition" [Doc. 3]. For more detailed references, See section "Related documents" of this document. These documents have to be seen as an integral part of this report!

Due to the tightly coupling between these documents, the framework version of "The evaluation test report definition" has to match the framework version of this evaluation report (which is 2.9). More information about the documents and tests versions together with their corresponding relation between both can be found in "The evaluation framework", see [Doc. 1] in section "Related documents" of this document.

The generic test code used to perform these tests can be downloaded on our website by using the link in the related documents section.

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 "Related documents").

1.3 Conventions

Throughout this document, we use certain typographical conventions to distinguish technical terms. Our used conventions are the following:

- ***Bold Italic*** for OS Objects
- **Bold** for Libraries, packets, directories, software, OSs...
- `Courier New` for system calls (APIs...)

Related documents

Those are the documents that are closely related to this document. They can all be downloaded using following link:
http://www.dedicated-systems.com/encyc/buyersguide/rtos/evaluations

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 What is a good RTOS?
This document presents the criteria that Dedicated Systems Experts use to give an operating system the label “Real-Time”. The evaluation tests are based upon the criteria defined in this document.
EVA-2.9-GEN-02

Doc. 3 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. 4 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. 5 Linux 2.6.33.7.2-RT30
This document presents the qualitative discussion of the OS
1EVA-2.9-OS-LNX-104 Issue: 1 May 13, 2011

2 Introduction

This chapter talks about: 1) the OS that we are going to test and evaluate, 2) the real time patch integrated in this OS to achieve some real time performance and behavior tests, 3) the library used for interaction between the testing applications and the kernel, 4) the hardware on which the under testing OS will be employed.

2.1 Overview

The evaluation project started in 1995 and as such accumulates a long experience with different (RT) OSs. Today more and more embedded systems are equipped with **Linux** solutions using more or less real-time patches. Different vendors like MontaVista, Windriver, and Lynuxworks have now **Linux** variants in their product portfolio.

Since the kernel version 2.4, a lot of improvements regarding real-time behavior found their way into the standard "**Vanilla**" kernel. There is a well maintained real-time patch available (both have their origins from Ingo Molnar) called ***RT_PREEMPT*** patch. Remark that some real-time features (like priority-inheritance ***mutexes***, introduced in version 2.6.18) are already in the **Vanilla** kernel.

We believed that it is the time to test this kernel by our standard real-time behavior evaluation framework and find out how well it behaves.

For this evaluation, we used ***buildroot*** as target development system. On 8 June 2011, the first ***µClibc*** version (0.9.32) was released containing the **Native POSIX Thread Library (NPTL)**. Finally, ***µClibc*** is supporting the task to be used in systems with time requirements and supports the priority inheritance mutex.

It is the first time we evaluate the ***RT_PREEMPT*** kernel with the ***µClibc*** library, and as expected, the behavior is now the same as on ***glibc***.

Remark that the ***RT_PREEMPT*** patch degrades throughput performance which means that it should be used only when your project has low latency requirements. This is normal and a fundamental rule in real-time software: latency improvements have a negative impact on throughput and vice versa. Some quick measurements using an NFS mount stressing network and disk showed a negative throughput impact between 5 and 10% by enabling the ***RT_PREEMPT*** patch!

2.2 Evaluated (RTOS) product

This section describes the OS that Dedicated Systems tested using their Evaluation Testing Suite, and the hardware on which this OS was running during the testing.

2.2.1 Software

The operating system OS that will be evaluated is **Vanilla Linux** 2.6.33.7 with real-time patch v30.This RT patch was the latest version officially released by OSADL (the Open Source Automation Development Lab) on December 21, 2010. Being as OSDAL's latest stable release was our main reason for testing this version. The RT patches can be found at http://www.kernel.org/pub/linux/kernel/projects/rt/.

The evaluation of this kernel version (2.6.33.7.2-rt30) was performed using several performance and behavior tests. The testing results are applicable only to this version as other versions may have other significant performance figures and behavior.

The library used between the testing applications and the kernel is the **µClibc** version 0.9.32 as mentioned before. This interfacing library is important because user applications (when using POSIX calls) can access the real-time features of the kernel only if this library supports them. Otherwise, direct system calls in user space applications are needed.

Further, the kernel was configured to use a high frequency timer source as clock generation and all power management was disabled. The test application was started from a RAM disk (tmpfs) and the real-time run away protection was disabled by setting `/proc/sys/kernel/sched_rt_runtime_us to -1`. All these precautions are required if real-time performance is your goal.

2.2.2 Hardware

We tested this Linux version on an ATOM-based platform with the following characteristics:

- Motherboard: Advantech SOM-6760, PCI bus at 33MHz, using the System Controller Hub US15W.
- CPU: Intel Atom Z530 1.6GHz 133MHz Front Side Bus.
  - 32KByte L1 Instruction Cache,
  - 24KByte L1 Write Back Data Cache,
  - 512KByte 8-way L2 Cache (which can be reduced up to zero in some processor sleep states)
  - 1 core with hyper-threading support (however hyper threading was disabled during this test).
- RAM: 512MB DDR2
- VMETRO PCI exerciser in PCI slot 3 (PCI interrupt level D, local bus interrupt level 10)
- VMETRO PBT-315 PCI analyser in PCI slot 4.
- External and CPU internal cache was enabled during the tests.

3 Evaluation results summary

Keep in mind that the tested and evaluated product is **Vanilla Linux** 2.6.33.7 with ***RT_PREEMPT*** patch v30. If correctly used and configured, the ***RT_PREEMPT*** **Linux** system has the internals to provide some real-time characteristics.

Compared with the traditional RTOS that supports also memory protection between processes, the worst case latencies in **Linux** ***RT_PREEMPT*** are still around 5 to 10 times slower (depending on the RTOS you compare with). Our study and measurements show that the latencies are inbound and therefore this **Linux** version may be labeled Real-Time.

TAKE CARE: Using a wrong driver or wrong configuration can destroy real-time behavior. You need to follow the detailed rules described in the relevant document (Doc 5).

3.1 Positive points

- No license fees
- Source code available
- Extensible

3.2 Negative points

- The real-time characteristics of the OS are present only when everything is configured and built correctly (and not for all drivers)
- GPL license is not completely free and investment is required to build a marketable system. For instance, though demo systems can be built quickly with Linux, the debugging, tuning and verification required to build a stable system ready for long-term use is much more difficult.
- Setting up a complete embedded target from scratch is a daunting task.

3.3 Ratings

For a description of the ratings, see [Doc. 3].

RTOS Architecture	0	6	10
OS Documentation	0	4	10
OS Configuration	0	6	10
Internet Components	0	10	10
Development Tools	0	6	10
Installation and BSP	0	4	10
Test Results	0	4	10
Support	0	N.A.	10

Although [Doc. 3] gives a description of the ratings, comparison with other reports on other OS should help you understand the scoring.

4 Test Results

Test Results	0				4						10

Compared with traditional RTOS supporting inter-process memory as well, the worst case latencies in **Linux *RT_PREEMPT*** *are around 5 to 10 times larger (depending on the RTOS you compare with). But they are still inbound and thus considered as real-time!*

However, using wrong configuration (at build and/or at runtime) or incorrect behaving drivers can still destroy the potential real-time behavior.

One of the items that are surely candidate to be improved is the clock tick interrupt duration.

4.1 Calibration system test (CAL)

"Calibration tests" are performed to calibrate the tracing overhead compared with the processing power of the platform. Such tests are important to understand the accuracy of the measurements done in scope of this report, and for measuring the processing power of the platform. This calibration permits comparison with the results on other platforms.

4.1.1 Tracing overhead (CAL-P-TRC)

"Tracing overhead test" calibrates the tracing system overhead. It is more related to the hardware than the OS because its aim is to correct the measured time values.

In the rest of the document, the tracing overhead is subtracted from the obtained results.

For tracing, a PCI analyzer on the PCI bus was used. By accessing a device on the PCI bus (PDrive) we can retrieve timing information on events without any call to the OS. Accessing the PCI bus takes some overhead of course; however, there isn't any jitter at all in the overhead of the trace which in turn does not generate much extra inaccuracies.

In general, the results in this report are correct to +/- 0.2 µseconds. Therefore the results shown in the tables are rounded to 0.1 microseconds/

4.1.1.1 Test results

Test	result
Average tracing overhead	270 nsec
minimum tracing overhead	270 nsec
maximum tracing overhead	270 nsec

4.1.2 CPU power (CAL-P-CPU)

The "CPU power" test calibrates the CPU performance and the memory bandwidth of the used platform. This test is measured in different situations, starting from the situation where code and data are cached, until the situation where neither code nor data are cached. With such different situation tests, the effects of the cache can be calculated.

We have been seriously reworking this test lately. The CPU test uses only one data address; The non-cached version is about 172KB in size (instructions), while the cached version uses a loop (a bit unrolled to have a small loop overhead but so it fits in the L1 I-cache and it uses only two data words). The instruction cache test is done twice:

- The instructions have not been mapped yet(leading to TLB exceptions and page faults)
- There will not be any page faults (TLB exceptions will still happen).

This gives us a "feeling" about the impact of page faults, even if the test software is launched from a RAM file system and uses `mlockall`.

Further, we divided the data cache tests into a read test (reading content of a large array in non-cached case, and read a small array in a loop in the cached case) and a write test. Remark that we flush the caches in between the tests.

This rework shows that a worst-case / best-case scenario can cause significant performance impacts, something that in reality will almost surely never be that large (or you should be able to run everything using only L1 caches).

Due to the rework, the impact of being/ or not being in the I-Cache has enlarged enormously compared with previous tests.

Remark that the results of such tests will depend also to a high extent on the cache organization:

- Number of ways
- Line size
- Number of address bits used for index
- Virtual or physical addresses used as index.

Further, we can adapt the test for CPU which has larger cache sizes as the arrays have to be larger than the cache size (across all levels).

4.1.2.1 Test results

The results for the evaluated platform are shown below:

Test	no cache	cached	cache effect
CPU test: first load.	132.9 us		
CPU test: I Cache effect	127.5 us	23.5 us	5.4
MEM write test	59.8 us	23.2 us	2.6
MEM read test	51.4 us	23.3 us	2.2
Average caching effect (CPU and MEM)			3.4

The results show similar behavior as on our standard evaluation platform the Pentium MMX 200MHz:

Test	no cache	cached	cache effect
CPU test: first load, page faults	1562 us	272.7 us	
CPU test: I Cache effect	1504 us	271.7 us	5.5
MEM write test	815.2 us	738.6 us	1.1
MEM read test	1355 us	817.1 us	1.7
Average caching effect (CPU and MEM)			2.8

The results show a behavior which is similar to the behavior that we had when we tested the same OS on our standard evaluation platform-the Pentium MMX 200MHz:

- Instruction cache has the most impact on performance (you need to get each instruction from RAM, while there are always less than one memory accesses for each instruction). Remark that this test is built on purpose and thus an extreme worst case that in practice will never occur (you will never have >32KB instructions without loops in real environments).
- Write seems not to be postponed, compared with the Pentium MMX which does postpone writes. As a result, un-cached writes have impact as well.
- Cache has more impact here on the ATOM than on the Pentium: it means that the speed factor between RAM and CPU has increased.
- The performance difference between both platforms is huge, surely when taking a look to cached memory accesses

Because of the serious impact caused by not having your instructions in the cache, you should take extra safety margins in real-time behavior (worst case is less predictable) on this platform.

4.2 Clock tests (CLK)

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

4.2.1 Operating system clock setting (CLK-B-CFG)

The "OS clock setting" test examines the setting of the clock tick period in the operating system. This test shows the default clock timing as they are set by the BSP and/ or the kernel.

As this kernel is configured for running at 1000Hz, we expect a clock interrupt each 1ms. The following table shows the test results.

Test	result
Test succeeded	Yes
Tested clock period	1ms
Clock period adaptable	YES (by kernel config)

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

The "clock tick processing duration" test examines the clock tick processing duration in the kernel. The test results are extremely important, as the clock interrupt will disturb all the other performed measurements. Using a tickles kernel will not even prevent this from happening (it will only lower the number of occurrences). The kernel under test was not using the tickles timer option.

The bottom line of the figures in section 4.2.2.2 represents the normal loop time of the test if no clock interrupt occurs during the test loop. The upper line is generated by the samples when a clock interrupt occurred during the loop. The difference between the two lines is the clock tick processing duration.

☹ A strange behavior was detected in this test. A clock time duration of about 22 µs was detected (measured), which is a bad measurement compared with the traditional RTOS (at least a

factor 5, up to a factor of 10). This clock duration will impact all other real-time behavior and measurements.

This was the main reason for doing this as an initial measurement.

Remark as well that on a x86 platform the BIOS can and will generate System Management Interrupts which will impact latency as well and cannot be prevented by the OS (this is true for all RTOS). In industrial used BIOS, it should be possible to minimize the number of SMI so only real critical stuff is handled using the lowest latency possible. In the traces, we see two such events. Remark that QNX replaces the BIOS on this platform avoiding the SMI impact.

After the test application was started, the first sample took longer duration due to cache misses.

4.2.2.1 Test results

Test	result
CLOCK_LOOP_COUNTER	10000
Normal busy loop time	18 µs
Busy loop time with clock interrupt	40 µs, worst case 48 µs
Clock interrupt duration	22 µs to 30µs.

4.2.2.2 Diagrams

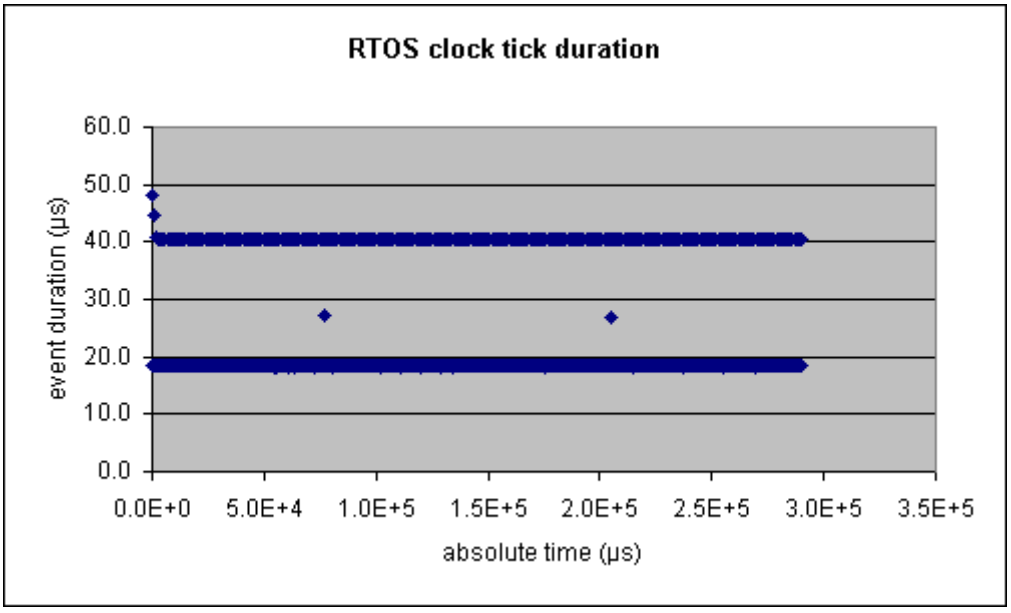


Figure 1: RTOS clock tick duration (1)

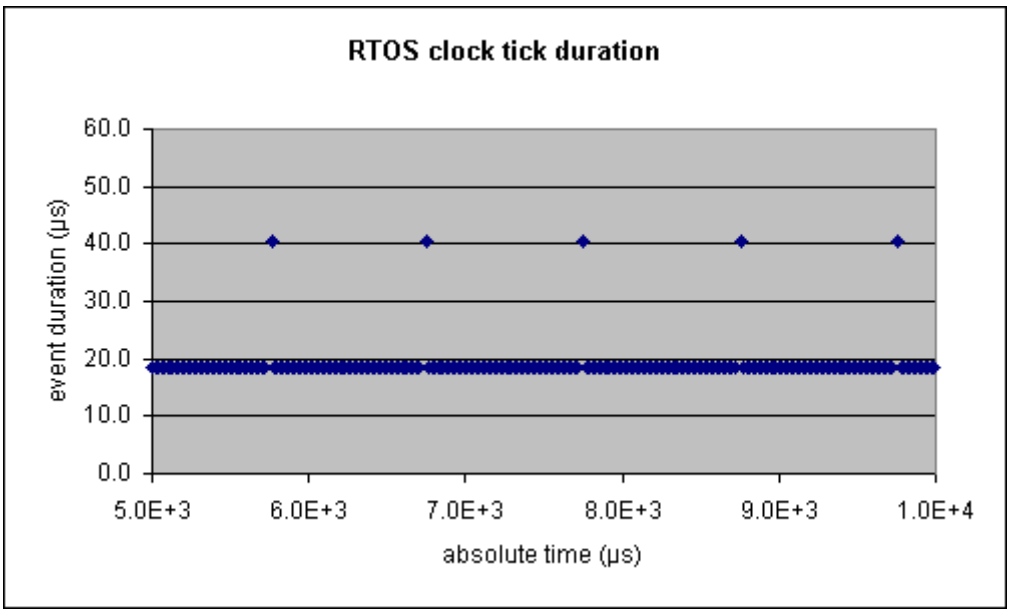


Figure 2: RTOS clock tick duration (2)

4.3 Thread tests (THR)

"Thread tests" measure the scheduler performance.

☹ Although most of the tests run pretty stable (deterministic), we found some strange issues:

- The queuing behavior on the ready queue while using ***SCHED_FIFO*** or ***SCHED_RR*** policy is different
- The first ***Round-Robin*** time slice of thread execution takes almost ten times more than the normal time slice.
- The first thread being deleted before it ever run in a process took consistently around 5ms, which is about 15 times longer than the other cases.

In real-time design, it is a bad practice to dynamically create and terminate threads, and to use multiple real-time threads on the same priority. Therefore, these problems should not popup in a good real-time OS designs but also shouldn't be avoided or ignored by real-time OS designers.

4.3.1 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?

This test succeeded.

However, 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 this is fundamental issue as it is not a good practice in real-time design to use the same priority for different threads. But it is still something to keep in mind.

4.3.1.1 Test results

Test	result
Test succeeded	YES
Lower priority not activated?	YES
Same priority at tail?	This depends if ***SCHED_FIFO*** or ***SCHED_RR*** scheduling class is used for the threads: RR puts it in front of its ready queue FIFO puts it at tail of its ready queue
Yielding works?	YES
Higher priority activated?	YES

4.3.2 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)!

☹ A problem was discovered here with the time slicing mechanism. When a thread is created (and thus put at the front of the ready queue), it seems to run for around 1s before giving the CPU back to the next thread in the ready queue with same priority.

Once all the threads are running, the time slice goes back to 10Hz frequency.

This can be clearly seen in the trace file when this test is done with 10 threads:

```
Sample     TimeAbs    TimeRel Data
  TRIG:      0.0ns      0.0ns F0000009
     1:   983.67ms   983.67ms F0000008
     2:     1.967s   983.67ms F0000007
     3:     2.951s   983.67ms F0000006
     4:     3.935s   983.67ms F0000005
     5:     4.918s   983.67ms F0000004
     6:     5.902s   983.67ms F0000003
     7:     6.886s   983.67ms F0000002
     8:     7.869s   983.67ms F0000001
     9:     8.853s   983.67ms F0000000
    10:     9.837s   983.67ms F0000009
    11:     9.936s   99.438ms F0000008
    12:     10.04s   99.438ms F0000007
    13:     10.14s   99.438ms F0000006
    14:     10.23s   99.438ms F0000005
```

Trace extract showing strange round robin behavior.

4.3.2.1 Test results

Test	result
Test succeeded	NO: first yield upon thread creation uses 983ms and next yields take 99ms!
RR Time slice following this test	100 ms

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

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 (on this platform, the L1 data cache is 24 KB and the instruction cache is 32 KB).

Further, you will clearly see the influence of clock interrupts (causing the maximum values in the graphics). As loading/starting the testing software passes a lot of code and data to the kernel, the next clock interrupt will not be cached (causing the 29µs delay for the first clock tick in the 10/128 thread scenario). Once there are enough running threads, the clock interrupt will always be un-

cached (at least data part of it) and thus for the 1000 thread tests, the clock interrupts always generate a delay of approximately 28µs.

For the rest, the thread switch latency is a fairly stable line (figures below), which is good.

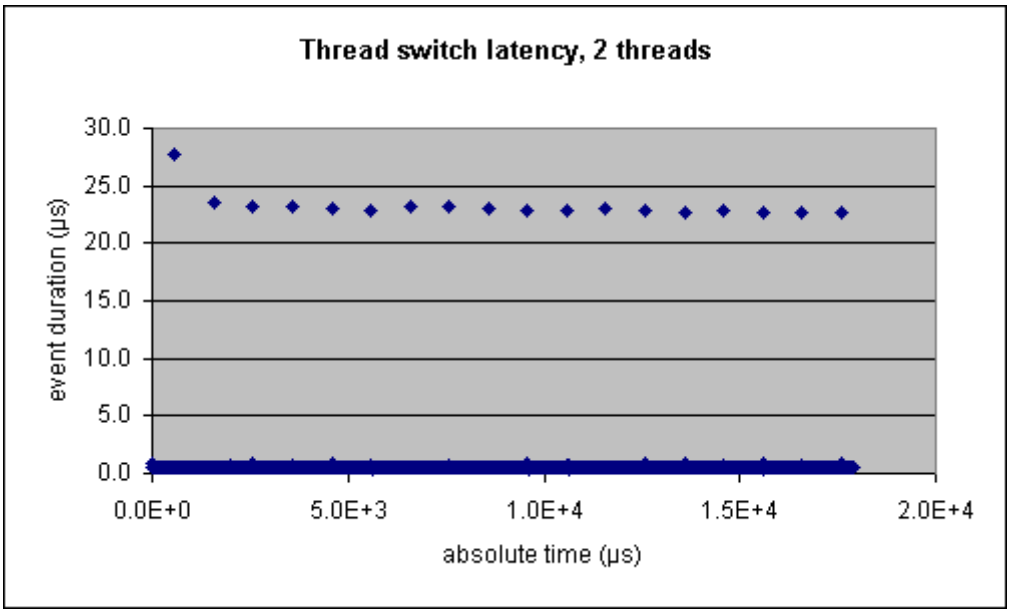
You see clearly here how bad the long duration of the clock tick is. This processor is very fast and therefore thread context switches take only a short time. But even with this fast CPU the clock tick stays high…

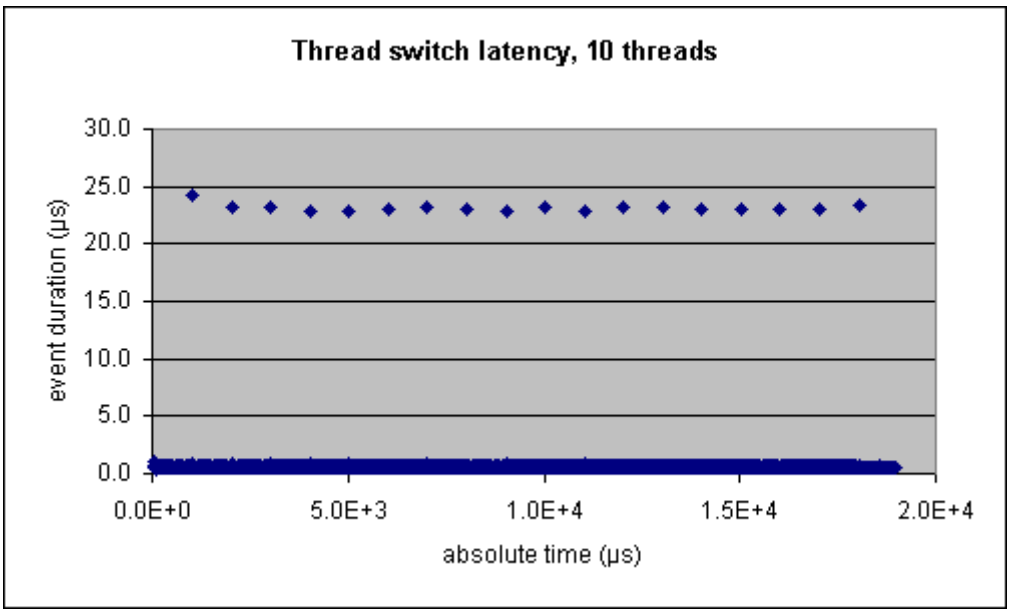
4.3.3.1 Test results

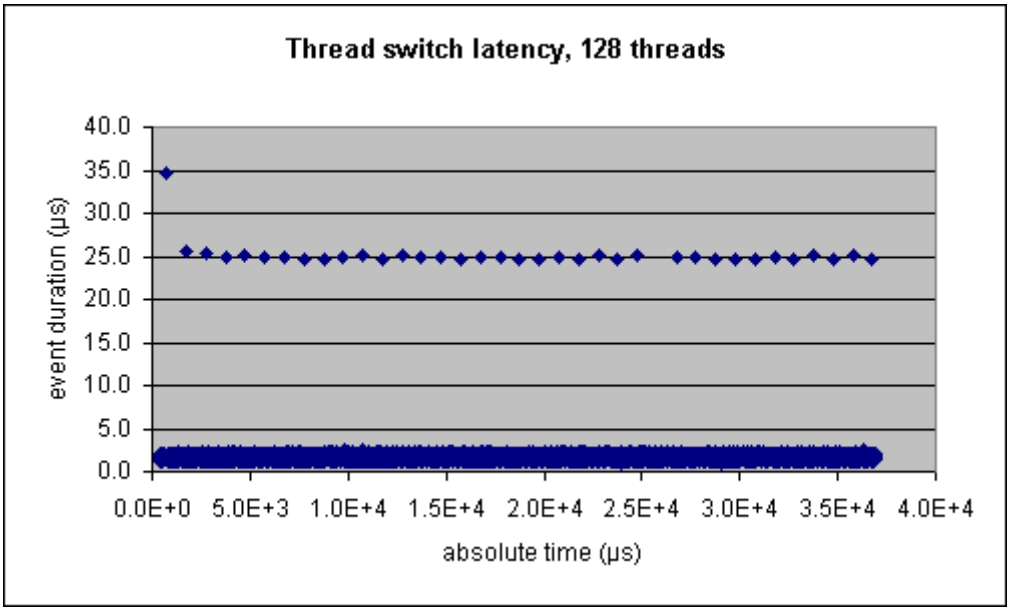
Test	result
Test succeeded	YES

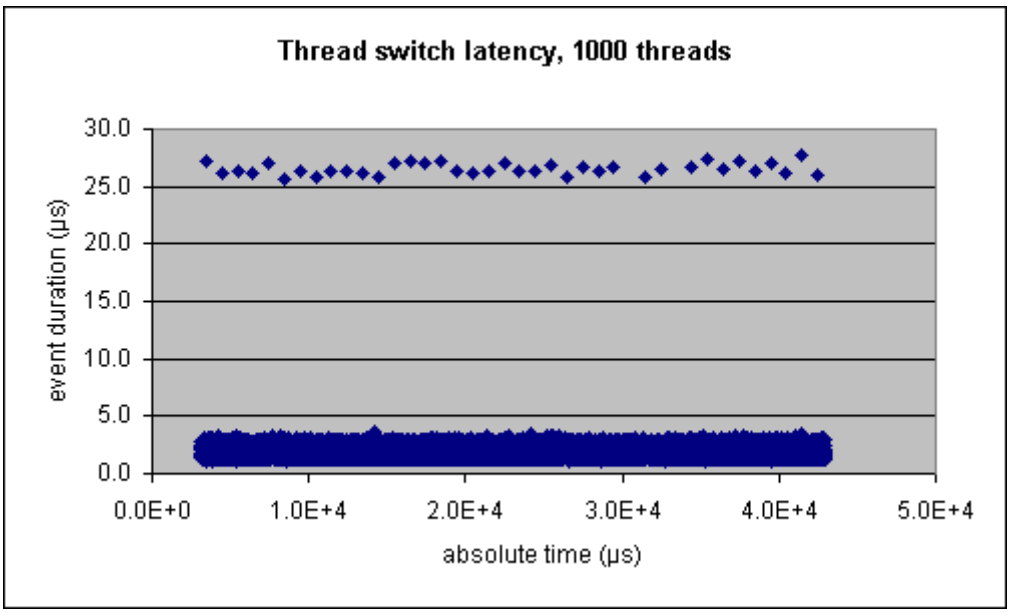
Test	Sample qty	Avg	Max	Min
Thread switch latency, 2 threads	16383	0.6 µs	27.7 µs	0.4 µs
Thread switch latency, 10 threads	16379	0.6 µs	24.2 µs	0.4 µs
Thread switch latency, 128 threads	16320	1.7 µs	34.6 µs	1.0 µs
Thread switch latency, 1000 threads	15884	2.0 µs	27.8 µs	1.1 µs

4.3.3.2 Diagrams









4.3.4 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).

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

We found on the x86 Pentium MMX platform a case that consistently took around 5 ms to handle: in the scenario of creating a thread and deleting it again before it ever run (scenario 1), the first thread requires always 5 ms to be deleted. Here the problem shows up again. Of course as the Atom is a much faster platform it takes in this case around 650µs.

Note: this problem was not present on the ARM even the same kernel source code and evaluation test code was used on these platforms. Therefore the problem has to be located in the architectural part of the kernel code.

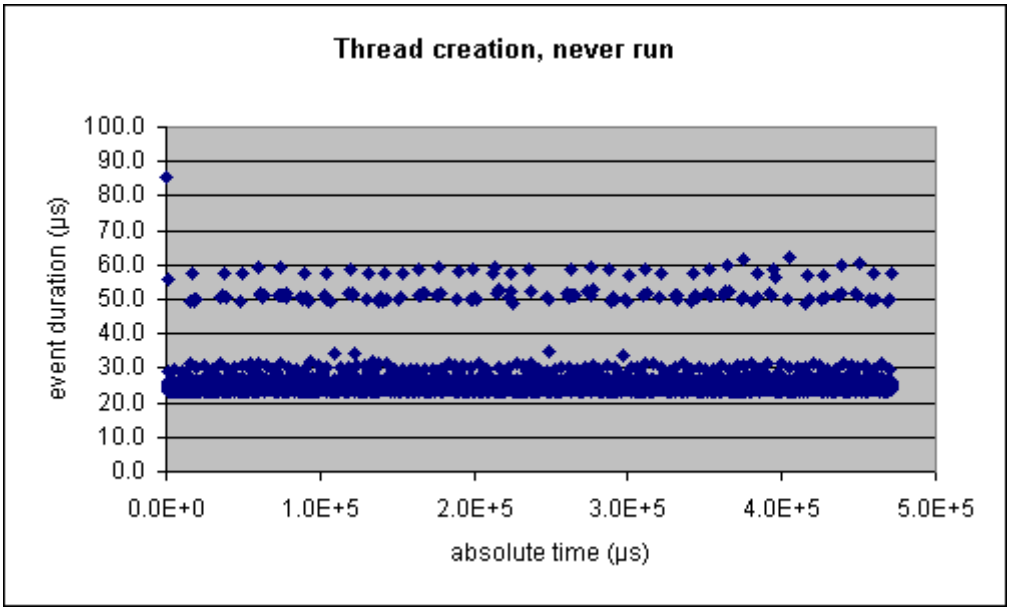
The clock tick is easily seen on the diagrams.

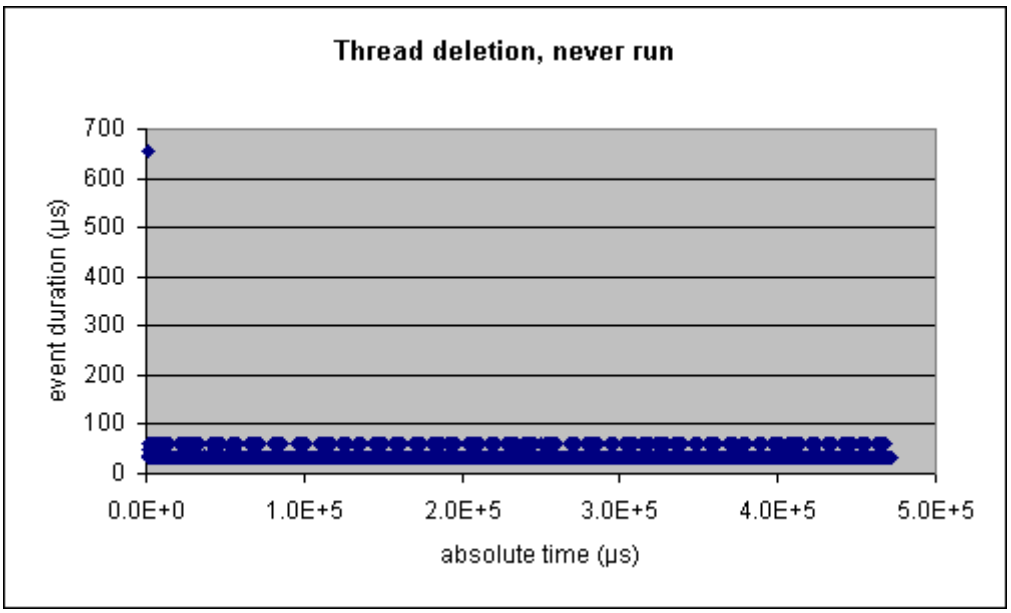
4.3.4.1 Test results

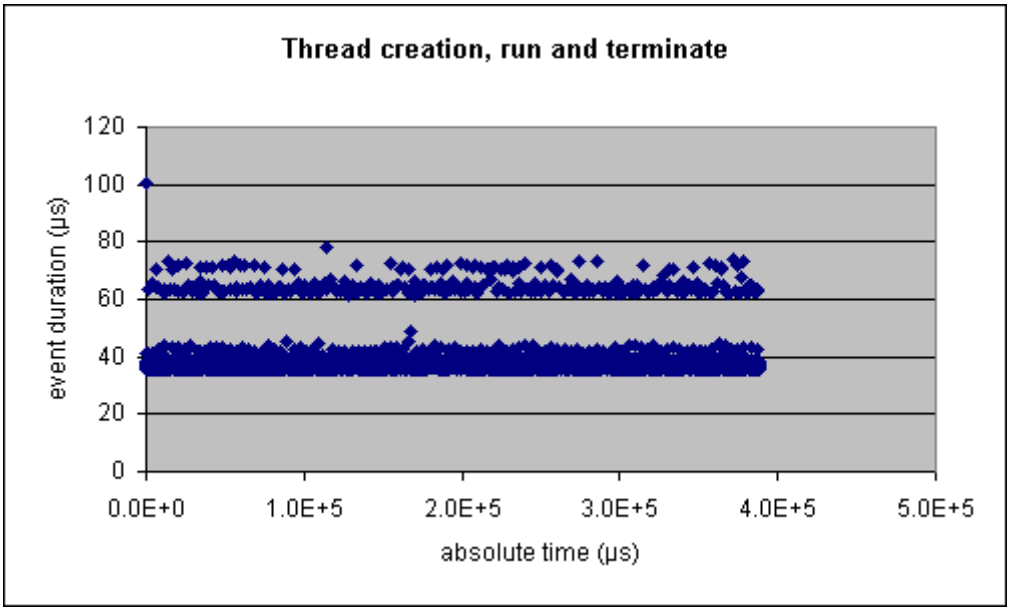
Test	result
Test succeeded	YES

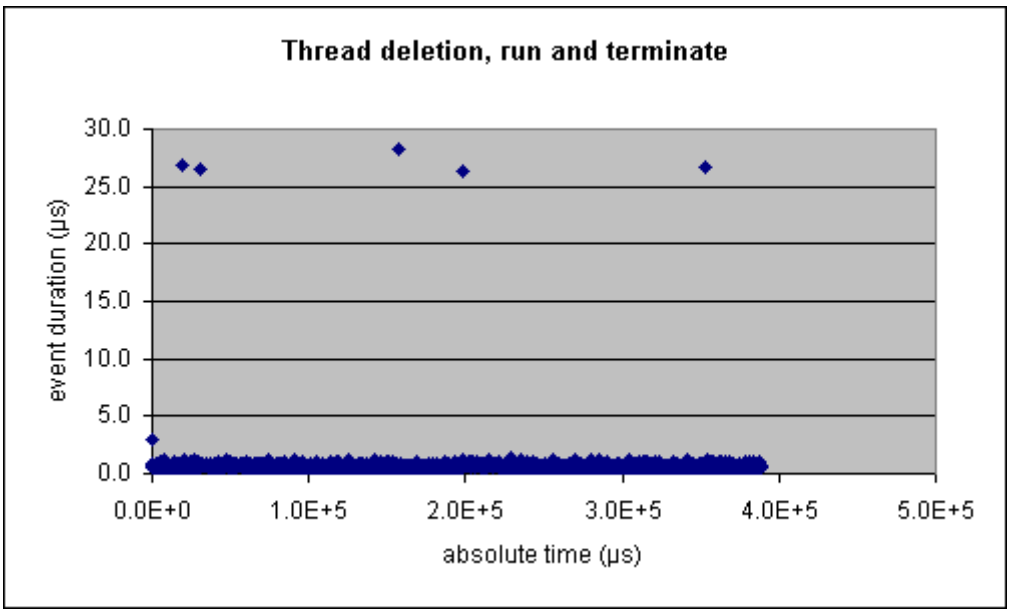
Test	Sample qty	Avg	Max	Min
Thread creation, never run	7500	25.3 µs	85.7 µs	23.1 µs
Thread deletion, never run	7500	34.8 µs	656.7 µs	33.1 µs
Thread creation, run and terminate	7500	38.1 µs	100.2 µs	35.4 µs
Thread deletion, run and terminate	7500	0.6 µs	28.2 µs	0.5 µs
Thread creation, run and block	7500	38.2 µs	100.4 µs	35.3 µs
Thread deletion, run and block	7500	20.3 µs	48.9 µs	16.7 µs

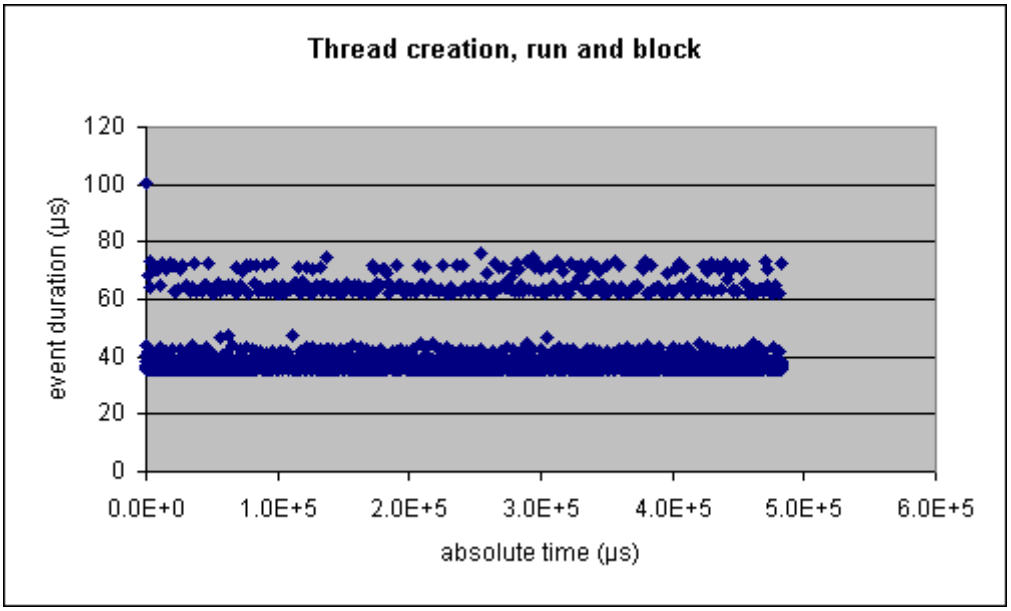
4.3.4.2 Diagrams

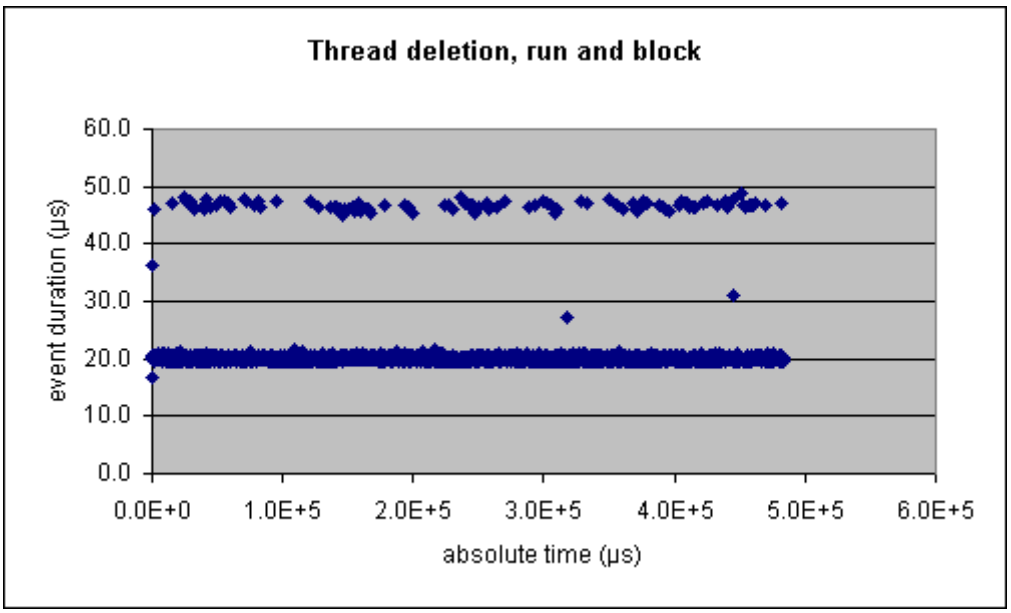












4.4 Semaphore tests (SEM)

"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.

4.4.1 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.

The semaphore behaves correctly as a protection mechanism.

4.4.1.1 Test results

Test	result
Test succeeded	YES
Maximum semaphore value?	Limited by the "int" type
Rescheduling on free?	OK

4.4.2 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.

This **Linux** version passed this test.

4.4.2.1 Test results

Test	result
Test succeeded	YES

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

The "time needed to create and delete a semaphore" test is performed to gain an insight about 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.

For a good RTOS, it is expected that there is no difference between the two scenarios. If a difference is detected, then this probably means that the OS handles some initializations on the ***semaphore*** on its first use (making the first use slower).

Anyhow, the tests show that no kernel interaction is needed to create/delete ***semaphores***. The duration of this time is just too small to be measured. The peaks we see are caused by clock interrupts.

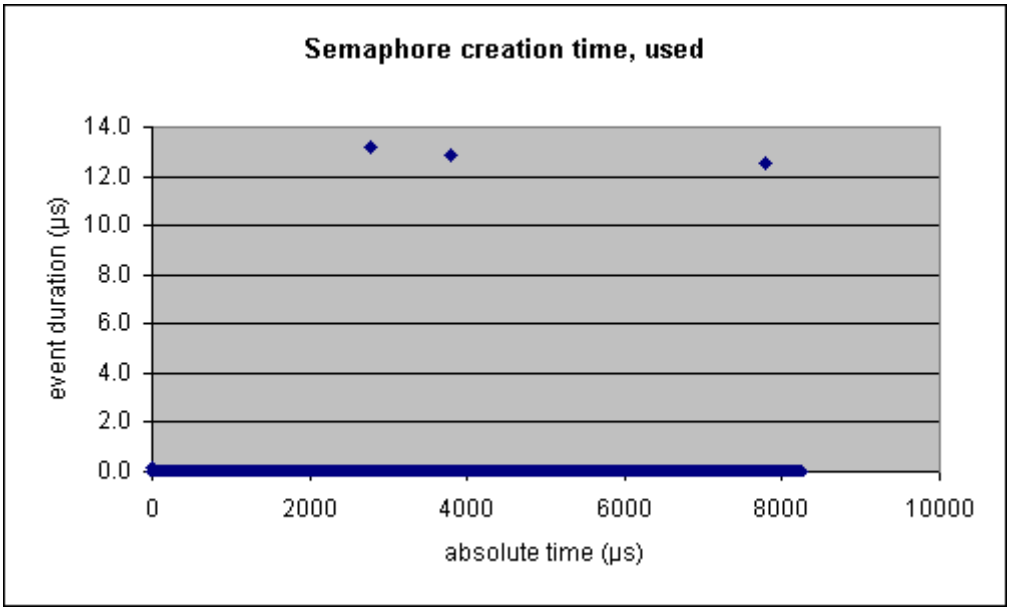
Remark that we do not use "named" ***semaphores***, so they cannot be used between processes. Therefore no access to the kernel is required.

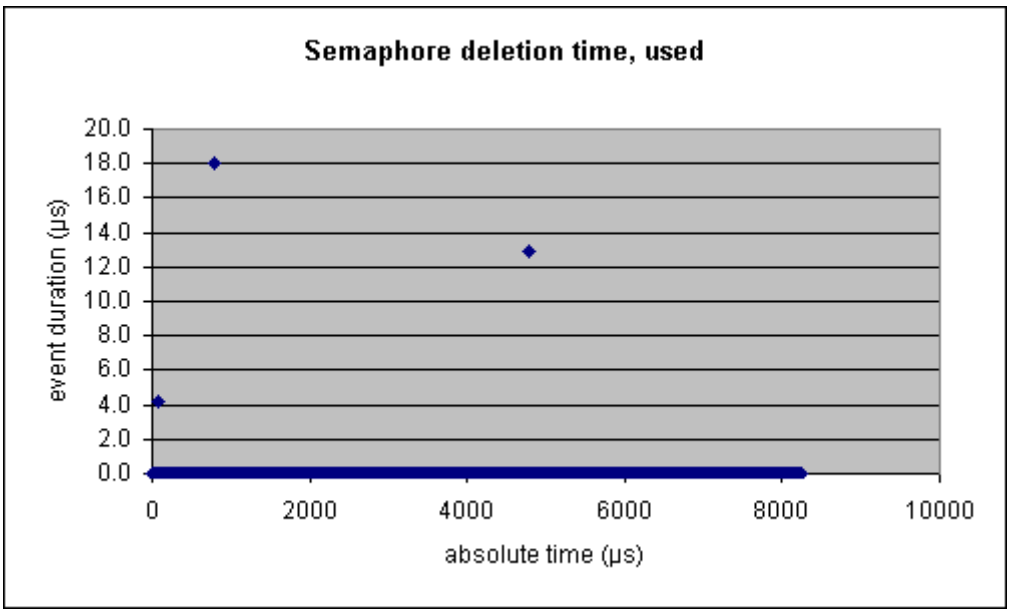
4.4.3.1 Test results

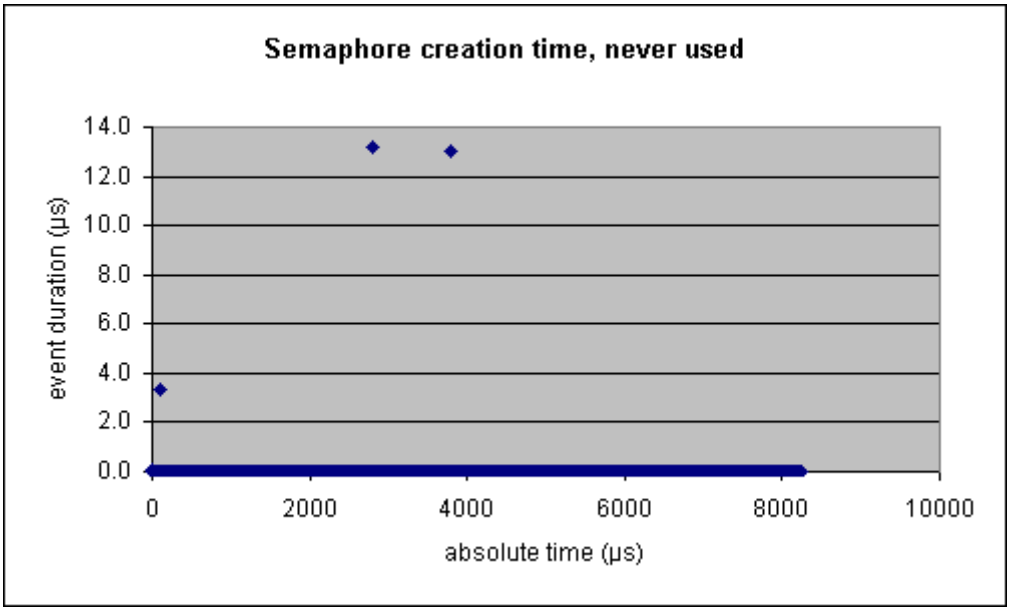
Test	result
Test succeeded	YES

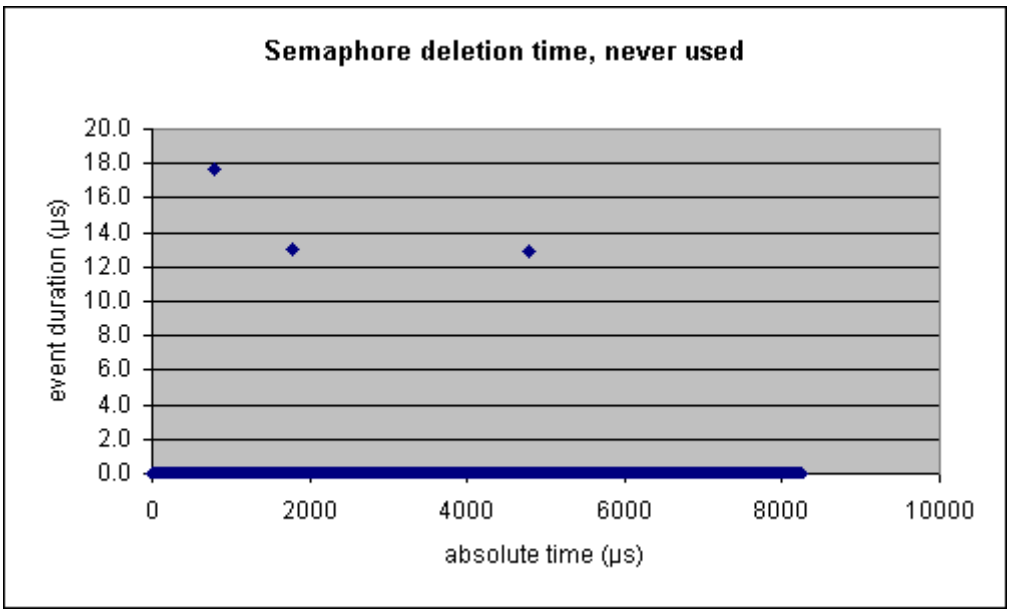
Test	Sample qty	Avg	Max	Min
Semaphore creation time, used	7500	<0.1 µs	13.2 µs	<0.1 µs
Semaphore deletion time, used	7500	<0.1 µs	18.0 µs	<0.1 µs
Semaphore creation time, never used	7500	<0.1 µs	13.2 µs	<0.1 µs
Semaphore deletion time, never used	7500	<0.1 µs	17.7 µs	<0.1 µs

4.4.3.2 Diagrams









4.4.4 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.

In fact, the library will only need to increase or decrease the ***semaphore*** counter in an atomic way (thus no system call involved). That's why it is too small to be measured.

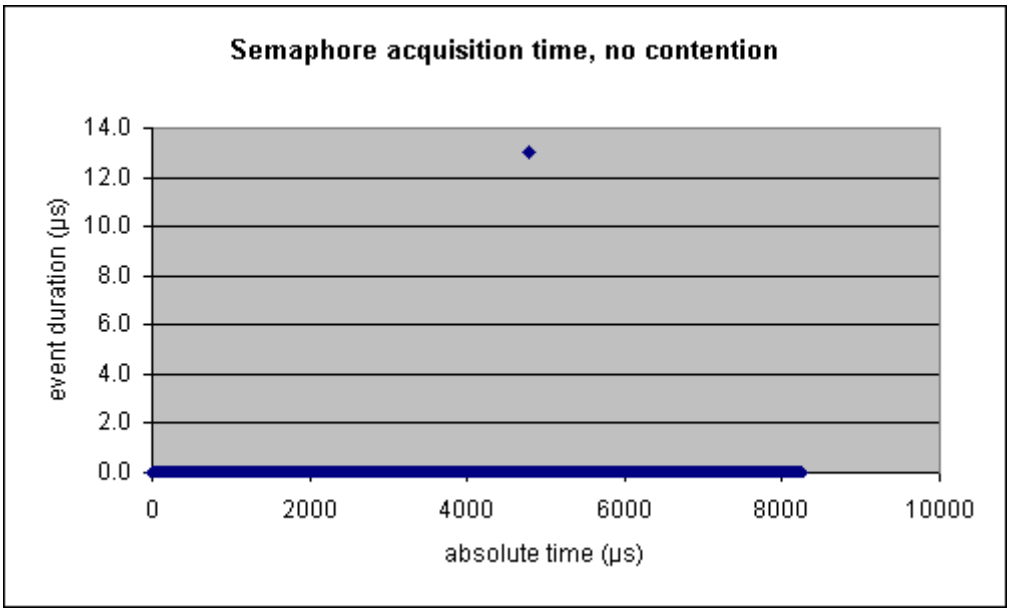
The clock tick spike is always present.

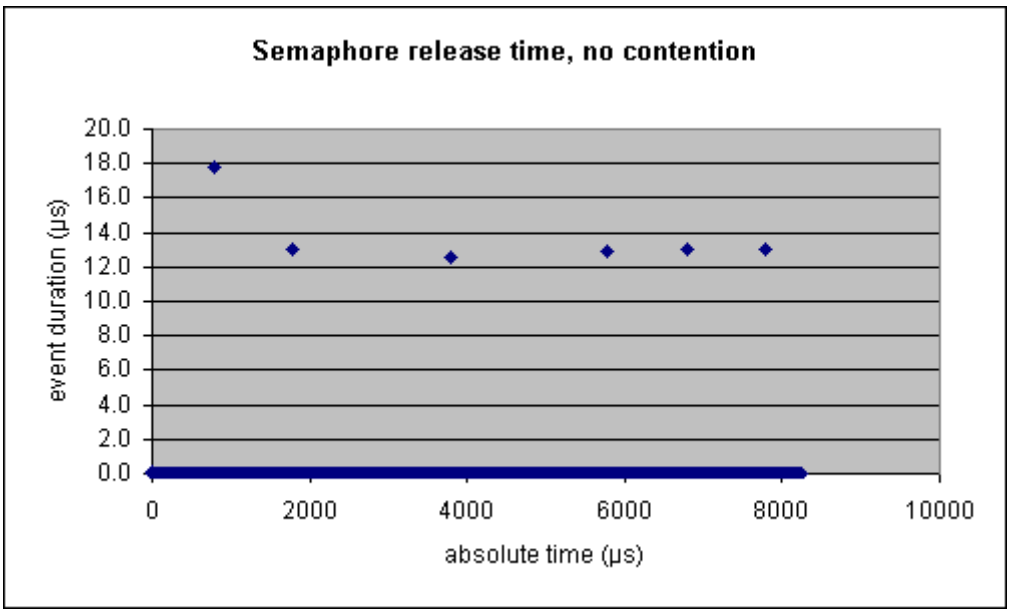
4.4.4.1 Test results

Test	result
Test succeeded	YES

Test	Sample qty	Avg	Max	Min
Semaphore acquisition time, no contention	7500	<0.1 µs	13.0 µs	<0.1 µs
Semaphore release time, no contention	7500	<0.1 µs	17.8 µs	<0.1 µs

4.4.4.2 Diagrams





4.4.5 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. To have a more clear view on our test, you can take a look on the expanded diagrams during a small time frame (e.g. one test loop):

- We create 90 threads with different priorities. The creating thread has a lower priority than the threads being created.
- When the thread starts execution, it tries to acquire the ***semaphore***; but as it is taken, the thread stops and the kernel switch back to the creating thread. The time from the acquisition attempt (which fails) for the moment the creating thread is activated again is called here the "acquisition time". Thus, this time includes the thread switch time.
- Thread creation takes some time, so the time between each measurement point is large.
- After the last thread is created and 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 at the moment the ***semaphore*** is released which in turn will activate the pending thread with the highest priority, which will stop the timing (thus again the thread switch time is included).

Now, the most important part of this test is to see if the number of threads pending on a ***semaphore*** has an impact on release times. Clearly, it doesn't, so this is good.

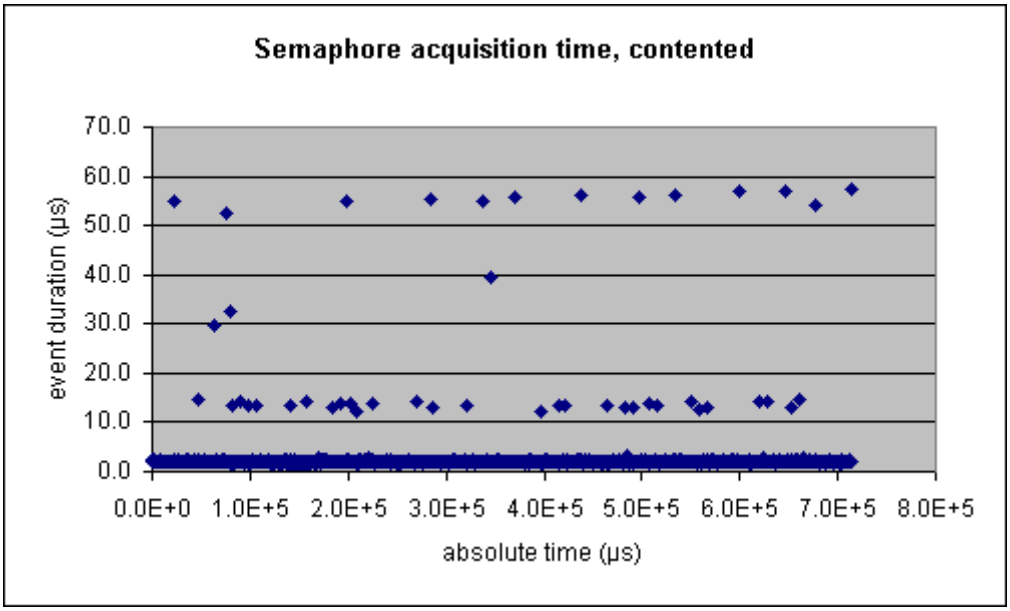
☹ As usual, we find the clock tick back in these diagrams; however, now we see something strange: it seems that the clock tick can loop a number of times, as we see clearly the distinct lines separated by clock tick duration (figures below in section 4.4.5.2). This is something strange… and it impacts badly real-time behavior! This is the same behavior as detected on other platforms, using this Linux version (even when using *glibc*)

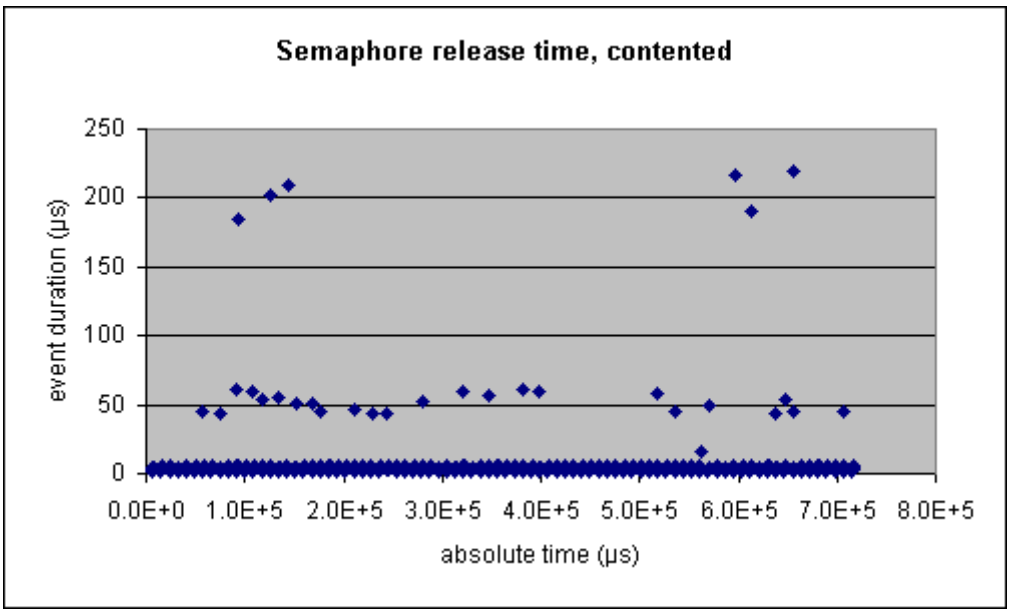
4.4.5.1 Test results

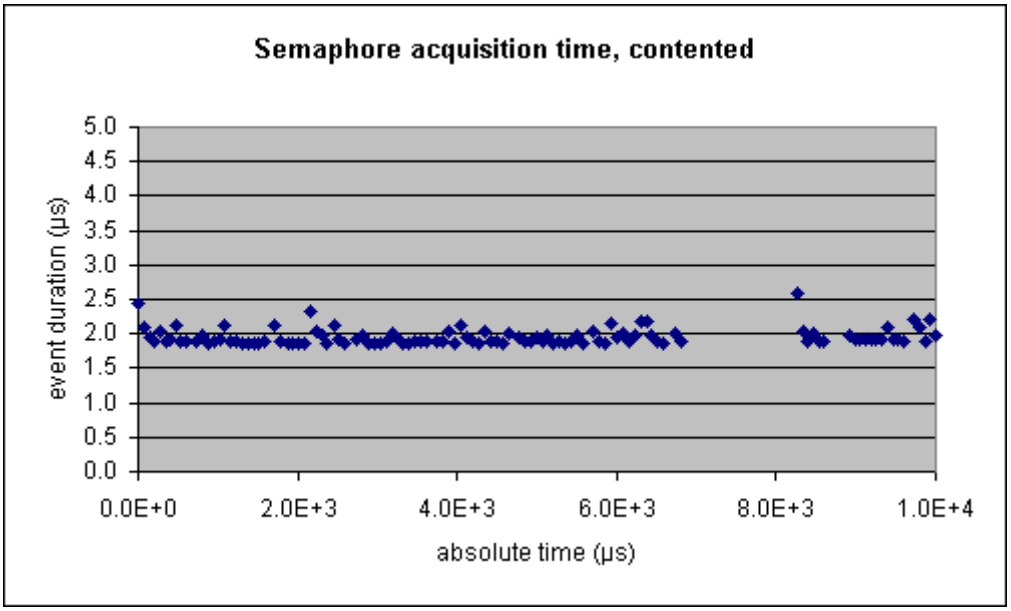
Test	result
Test succeeded	YES
Max number of threads pending	90 (close to the number of priority levels)

Test	Sample qty	Avg	Max	Min
Semaphore acquisition time, contented	7476	2.1 µs	57.6 µs	1.8 µs
Semaphore release time, contented	7476	4.4 µs	219 µs	1.8 µs

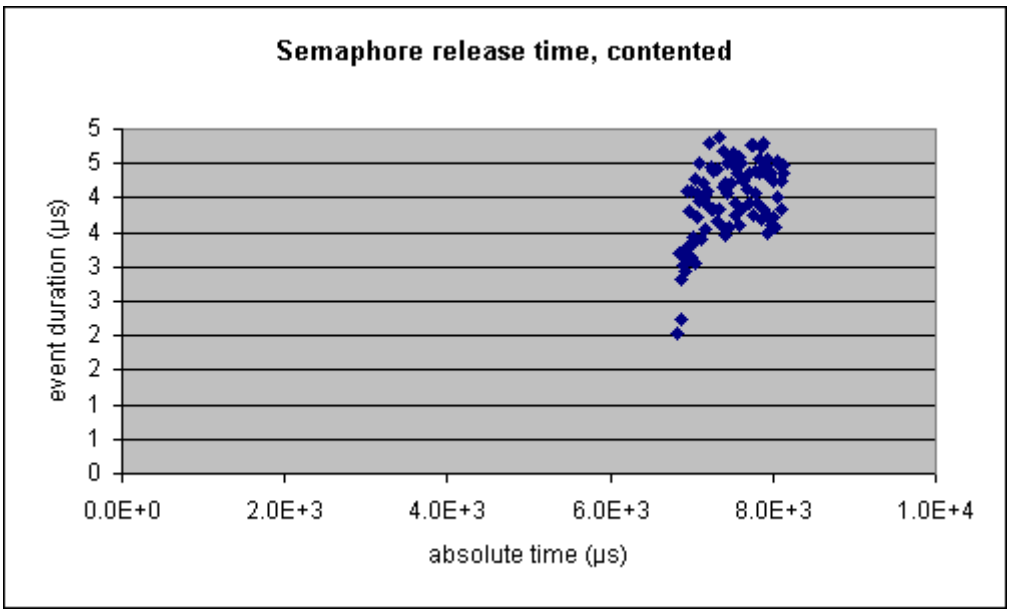
4.4.5.2 Diagrams







Zoomed in extract



Zoomed in extract

4.5 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.

4.5.1 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.

As this is one of the first ***RT_PREEMPT*** achievements going into the mainstream kernel, we did not expect any problems. Priority inversion behaves as expected.

4.5.1.1 Test results

Test	result
Priority inversion avoidance system call present	Yes
System call used	pthread_mutex_lock
Test succeeded	YES
Priority inversion avoided	YES
Mechanism used if any?	pthread_mutexattr_setprotocol: PTHREAD_PRIO_INHERIT

4.5.2 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" 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.

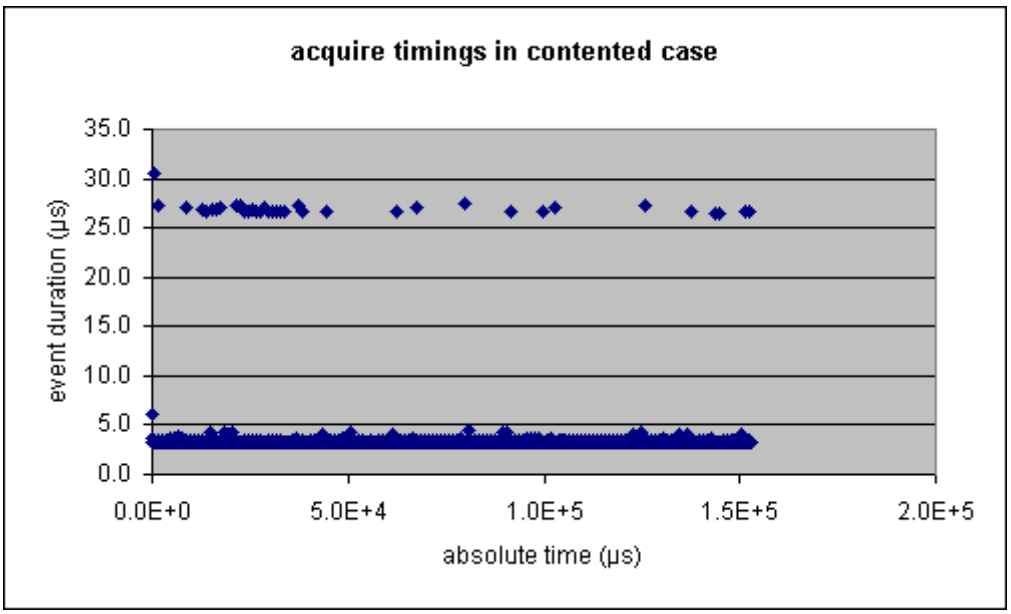
☹ We see that the release seem to cause a double thread switch (time take double as long), which is very strange! None of the other tested RTOS had such behavior.

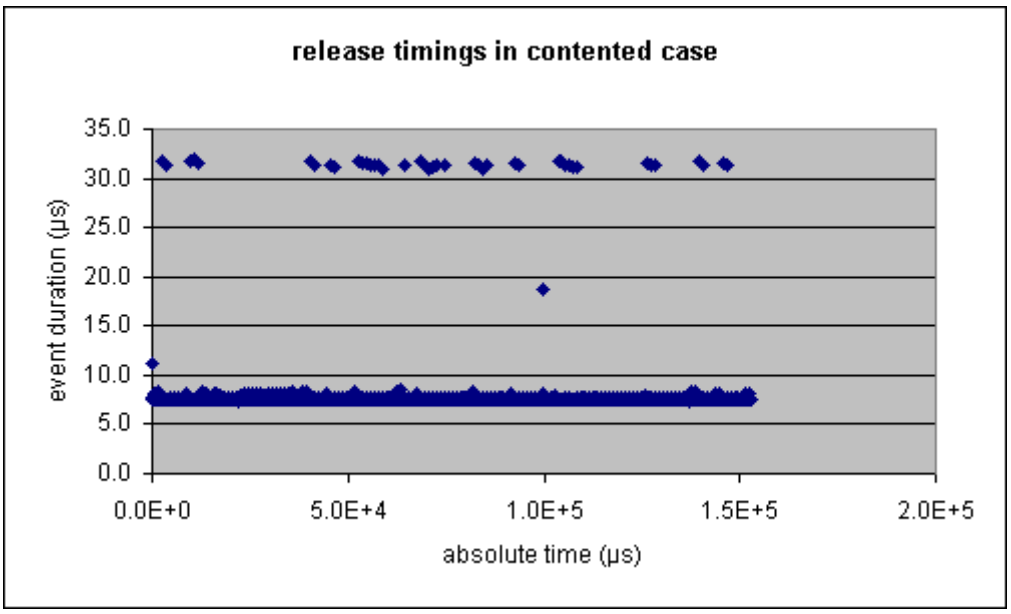
4.5.2.1 Test results

Test	result
Test succeeded	Yes

Test	Sample qty	Avg	Max	Min
Mutex acquisition time, contention	7500	3.4 µs	30.5 µs	3.2 µs
Mutex release time, contention	7500	7.7 µs	31.2 µs	7.4 µs

4.5.2.2 Diagrams:





4.5.3 Mutex acquire-release timings: non-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.

We have seen **Linux** running on embedded CPUs that do not have an atomic instruction, which in this case requires a call to the kernel! On such platforms, the performance of multi-threaded application which requires locks is seriously impacted! We observed once an extreme case, where approximately half of the CPU load was caused by the lock-need-to-go-to-the-kernel calls!

As expected, this doesn't take a lot of time, although it takes more time than a ***semaphore***. Maybe the default ***mutex*** attribute is not the FAST one? A ***mutex*** can be configured with different options: recursive, error checking and so one, which can increase the time required to take/release a lock.

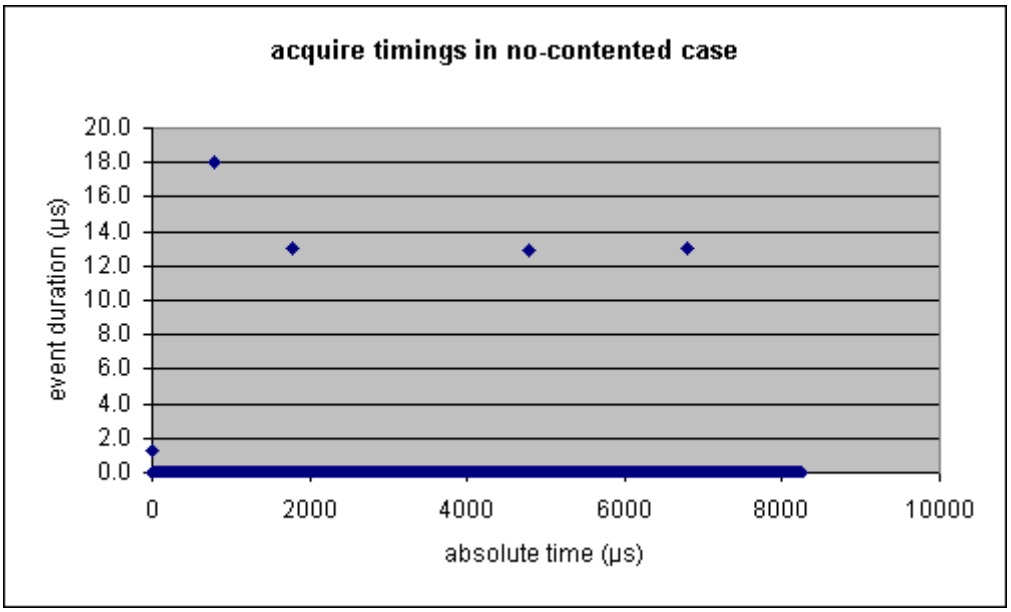
As in all diagrams, the clock tick shows up again.

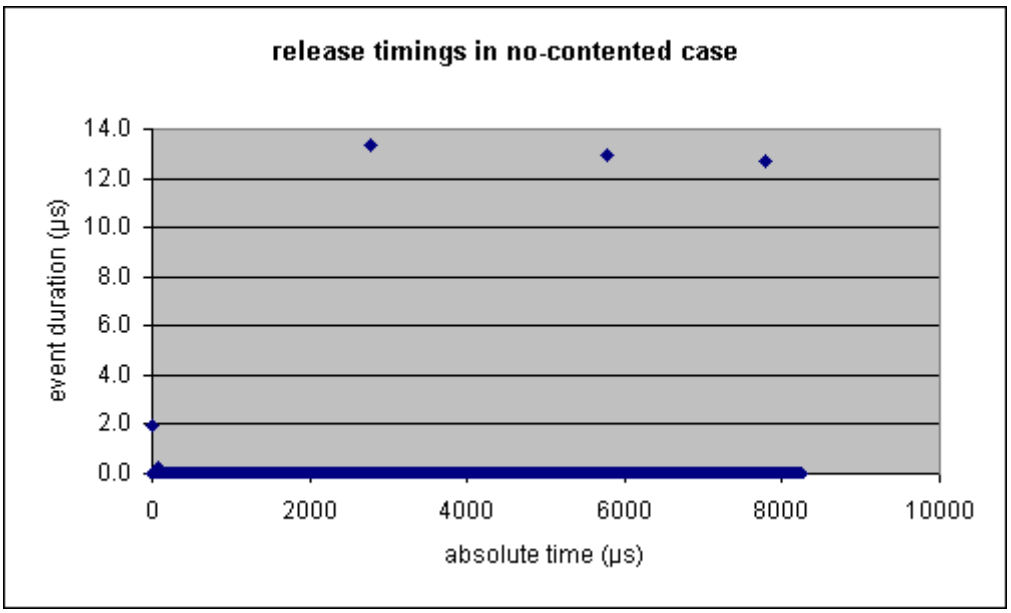
4.5.3.1 Test results

Test	result
Test succeeded	Yes

Test	Sample qty	Avg	Max	Min
mutex acquisition time, no contention	7500	<0.1 µs	18.1 µs	<0.1 µs
mutex release time, no contention	7500	<0.1 µs	16.5 µs	<0.1 µs

4.5.3.2 Diagrams:





4.6 Interrupt tests (IRQ)

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

Interrupt handling is a key system capability of real-time operating systems. Indeed, RTOSs are typically event driven.

On this platform, we use a PCI device which generates interrupts using an internal timer (thus independent of operating system clock being tested).

On this platform, some drivers shared interrupts with our device. These drivers use `hardirq` threads, but as a result it was not possible to directly use the interrupt context in our test scenario. We solved this by powering down the PCI slots that shared the interrupt, so we had the interrupt for our test only.

Remark that we measure here the best case: we use a real interrupt while the other interrupts use `hardirq` thread. Adding an interrupt to thread switch latency will then give you interrupt latencies to the `hardirq` threads.

So the results are showing the delay from the hardware interrupt line towards the interrupt context (like the other operating systems tested on this platform). Remark that SMI (System Management Interrupts) from the BIOS can also interfere. This also applies for the other operating systems tested on this platform.

4.6.1 Interrupt latency (IRQ_P_LAT)

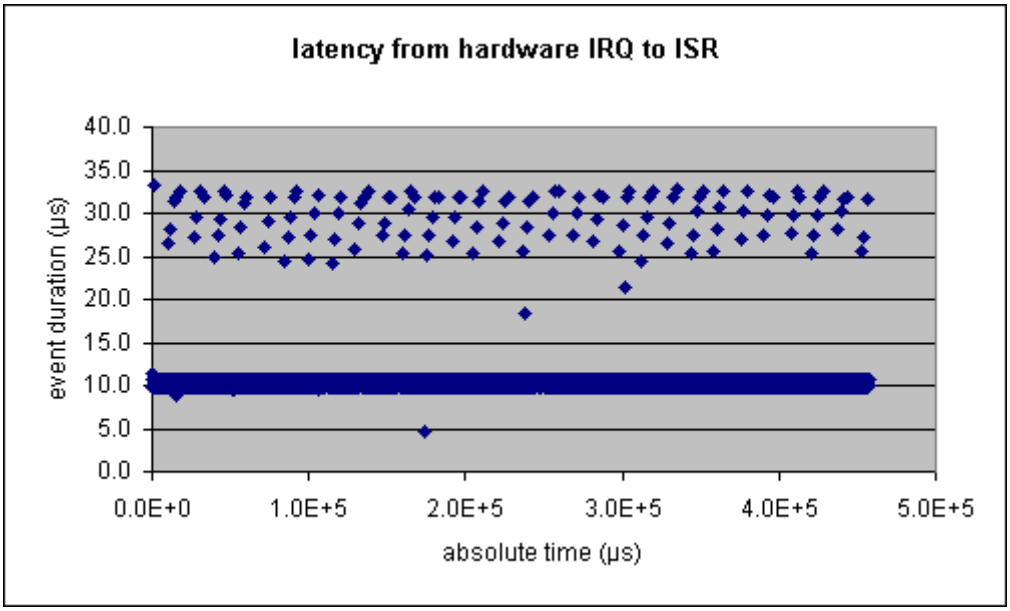
The "interrupt latency" test measures the time it takes from the hardware interrupt going high until the interrupt handler clearing it. So this measurement does also takes into account the hardware interrupt latency.

Clearly the timer interrupt impact is seen. At one location we had a very short latency; taking a look into the analyzer traces showed us that at that time an interrupt was almost missed. So the interrupt handler directly entered again before the kernel could switch back to the interrupted thread.

4.6.1.1 Test results

Test	Sample qty	Avg	Max	Min
Interrupt dispatch latency	10922	10.4 µs	33.4 µs	4.6 µs

4.6.1.2 Diagrams



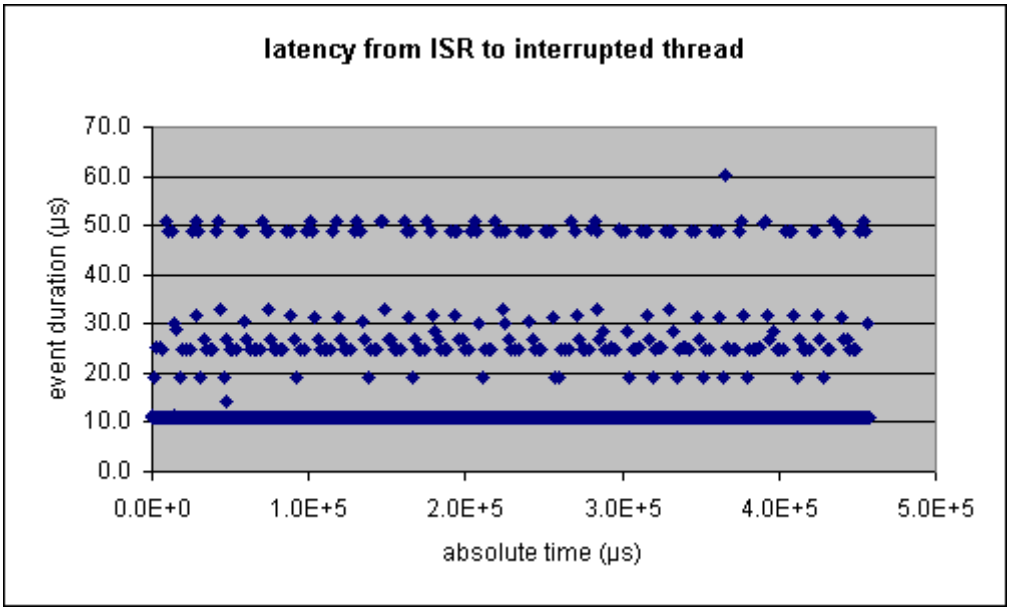
4.6.2 Interrupt dispatch latency (IRQ_P_DLT)

The "interrupt dispatch latency" test measures the time the OS takes to switch from the interrupt handler back to the interrupted thread.

4.6.2.1 Test results

Test	Sample qty	Avg	Max	Min
Dispatch latency from interrupt handler	10922	11.5µs	60.2µs	11.0µs

4.6.2.2 Diagrams



4.6.3 Interrupt to thread latency (IRQ_P_TLT)

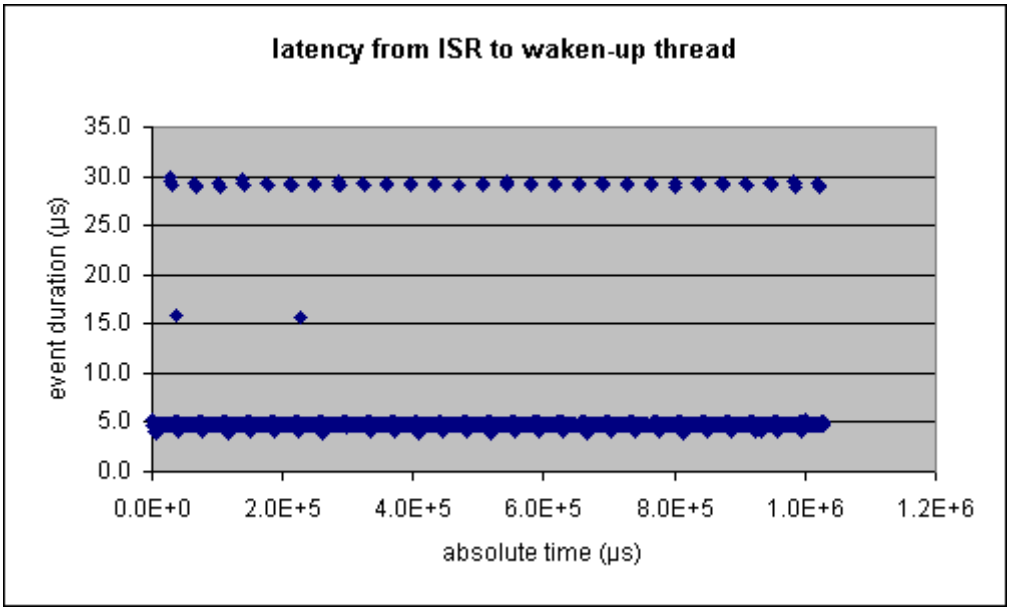
The "interrupt to thread latency" test measures the time the OS takes to switch from the interrupt handler to the thread that is activated from the interrupt handler.

This test is done by allowing the interrupt handler to release a blocked thread. This blocking thread has the highest priority in the system and is blocked by an `ioctl` waiting for the next interrupt handled by our test device driver. There is also a low priority thread looping. So the measurement takes the time from the interrupt handler to the blocked thread (as a consequence this includes a thread switch).

4.6.3.1 Test results

Test	Sample qty	Avg	Max	Min
Latency from ISR to waken-up thread	8132	4.9µs	29.9µs	3.9µs

4.6.3.2 Diagrams



4.6.4 Maximum sustained interrupt frequency (IRQ_S_SUS)

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

This test is done in 3 phases:

- 1000 interrupts as an initial phase: a fast test just to see where we have to start searching.
- 1 000 000 interrupts as a second phase based on the results from the first phase. This test still takes less than a minute and gives already accurate results.
- 1 billion interrupts as a last phase, which takes few hours and sometimes more than 24 hours, depending on the used platform and OS. This phase is done to verify stability; therefore, we cannot run this phase many times, especially when it comes to large interrupt latencies.

As one can observe in the test results, although the interrupt latency is in the best case only a 36.8 µs, the clock tick gives us a penalty here. On the long run, you can see that the guaranteed interrupt latency comes around 47.4 µs.

4.6.4.1 Test results

Interrupt period	#interrupts generated	#interrupts lost
34 µs	1000	5
34.9 µs	1000	3
35.3 µs	1000	2
36.8µs	1000	0
36.8 µs	10,000	0
36.8 µs	100,000	0
36.8 µs	1 million	6
38.3 µs	1 million	4
39.7 µs	1 million	1
40.6 µs	1 million	1
42.6 µs	1 million	0
42.6 µs	10 million	12
44.5 µs	10 million	5
46.4 µs	10 million	0
46.4 µs	100 million	0
47.4 µs	1 billion	0

5 Support

Support	0					NA					10

If you use downloaded open source software, you have only the internet as documentation resource available.

Although a lot of information can be found on the internet, concerning kernel configuration and RT_PREEMPT it is much more difficult to find adequate and complete documentation.

6 Appendix B: 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