Performance Analysis of VxWorks and RTLinux

Benjamin Ip
COMS W4995-2, Languages of Embedded Systems
Department of Computer Science, Columbia University, NY

1. Abstract

This paper compares and evaluates the suitability of two real-time operating systems, the commercially available VxWorks and the publicly available RTLinux. Holding the hardware constant and using different measurement methodologies, we measured the overheads incurred during operating systems context switching, interrupt processing, object synchronization, and message passing. We also examine their effectiveness in terms of how they handle priority inversion problem.

Our finding illustrates that both VxWorks and RTLinux provide good raw performance. However, VxWorks is more predictable and deterministic, thereby making it more suitable as an operating system platform for developing and running soft and hard real-time applications.

2. Introduction

2.1. Overview of VxWorks

VxWorks is by far the most widely adopted commercial RTOS in the embedded industry. It is developed by WindRiver with the intention to design an operating system with fast, efficient, and deterministic context switching. Its *Wind* micro-kernel can support preemptive and round robin scheduling policies, and unlimited number of tasks with a maximum of 256 priority levels. VxWorks is also well known for its rich tool chain and run time library that significantly reduce the amount of time for application development. Despite the extensive features from VxWorks, it bares a high premium for royalty fee.

2.2. Overview of RTLinux

Unlike Linux, RTLinux provides hard real-time capability. It has a hybrid kernel architecture with a small real-time kernel coexists with the Linux kernel running as the lowest priority task. This combination allows RTLinux to provide highly optimized, time-shared services in parallel with the real-time, predictable, and low-latency execution. Besides this unique feature, RTLinux is freely available to the public[1]. As more development tools are geared towards RTLinux, it will become a dominant player in the embedded market.

3. Performance Metrics

Our project goal is to study the performance analysis of these two operating systems by measuring the following key metrics.

3.1. Context Switch

Both RTOSes are designed to support multitasking. This feature is important for real-time applications that are frequently implemented with multiple asynchronous tasks of execution. During task scheduling, a context switch is needed to suspend one task and immediately resume the other. Therefore, it is fundamental to analyze the average context switch latency in order to measure operating systems responsiveness.

3.2. Priority Inversion

Priority inversion occurs when a high-priority task is blocked, waiting for a low-priority task to release a resource shared by the high priority task. Priority inversion is a serious problem in real-time system since it often leads to deadlock. Both RTOSes incorporate their own priority inheritance protocol and one of the project goals is to examine the effectiveness of these protocols.

3.3. Interrupt Latency

Interrupt Latency is defined as the sum of interrupt blocking time during which the kernel is pending to respond to an interrupt, saving the tasks context, determining the interrupt source, and invoking the interrupt handler. For a particular interrupt, the latency also includes the execution time of other nested interrupt handlers. Since most embedded systems are interrupt-

[1] RTLinux is distributed by Finite State Machine

driven, low interrupt latency will drastically increase system throughput.

3.4. Synchronization

A full suite of synchronization methods is provided by VxWorks and RTLinux to allow exclusive access of shared resources. Acquiring and releasing semaphores to protect shared objects do incur penalty. Such penalty is often associated with adding and removing the requested tasks into and out of the object lock queues. As such, measuring synchronization overhead is another way to determine the viability of a real-time operating system.

3.5. Inter-Process Communications

Modern real-time applications are constructed as a set of independent, cooperative tasks. Along with high-speed semaphores, VxWorks and RTLinux also provide message queue as higher-level synchronization mechanism to allow cooperating tasks to communicate with each other. Because of the implementation complexity, using this service imposes the greatest amount of latency and thus is a key metric to operating system study.

3.6. Measurement Process

Measuring the above metric requires certain degree of resolution, accuracy, and granularity. Throughout our project, both hardware and software logic analyzers were used to capture and record measurement samples. We emphasized on the use of the hardware logic analyzer because it gives the finest resolution, least obtrusion to real-time code, and more important it is platform independent. In most test cases the software analyzer was used for verification. We also developed small firmware code to setup the memory maps and interrupt vector tables, tune the system clock, and disable the hardware cache. All test functions and system calls written to initialize tasks, semaphores and message queues are POSIX compliant. Finally, each test was measured with a sample size of 25 to ensure that the data collected are statistically sound.

3.7. Test Environment

Our tests are conducted under VxWorks version 5.4 and RTLinux version 3.0. We executed all of our tests on evaluation boards manufactured by FSM and WindRiver. Each evaluation board comes with a single MPC8260 microprocessor and its board support package. These significantly reduce development time spent on configuring the hardware.

4. Related Work

Performance Analysis of operating systems has long been an interesting subject among research groups. Levine [1] and his peers have presented their benchmark results on context switch time and priority inversion protocol latency of a real-time CORBA[2] architecture. Levine's [1] method of detecting and observing priority inversion is complicated. A straightforward way to create a priority inversion scenario is explained in Obenland's [4] article and will be described in next section.

Sohal [3] took both the analytical and empirical approaches to measure different phases of interrupt latency of a real-time operating system. Due to time constraint, we chose only one of Sohal's interrupt tests that typically reveals the performance of interrupt handling. However, we did not apply Sun's [4] approach to measure interrupt latency because Linux interrupt mechanism is implemented vastly different from RTLinux (with no distinction of top and bottom half of interrupt service routine).

We also learnt from Obenland's [2] experience that prior to executing any IPC test, the message queue should have no message pending and the receiving task must be blocked waiting for the message.

In Stewards [4] paper, he devoted much of his time to review and explain various measurement techniques that produce results of different granularity. We are convinced by Stewards [4] that hardware logic analyzer is preferable to other tools and techniques for measurements throughout the project.

5. Test Methods and Experimental Results

We modified some of the test methods referenced in the previous section to achieve fair and accurate results. This section describes the measurement outcomes along with the methods that we used to test different metrics.

[2] Common Open Brokerage Architecture

5.1. Context Switch

We configured both RTOSes to use round-robin scheduling policy to determine context switch time. Figure 1 shows that with round-robin policy, we simply need to create two tasks and let the scheduler to execute them alternately, without the need of prioritizing them.

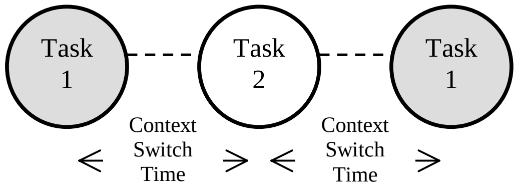

Figure 1: Context Switch Test Setup

Both tasks under test have the same function; each contains an infinite empty loop to avoid additional computation. Table 1 shows the average (and standard deviation) context switch time measured in microseconds.

	VxWorks	RTLinux
Context Switch (µS) Mean (Std)	11 (0.04)	13.4 (0.6)

Table 1: Context Switch Time Measurements

The context switch time measured on VxWorks is consistently low, with a standard deviation of 0.04. On the contrary, the RTLinux context switch time is 18% higher than and it is not as consistent (with std of 0.6) as VxWorks. Thus, context switch time for VxWorks is more deterministic. The lower score achieved by RTLinux seems to imply that running both real-time and non real-time tasks in parallel may not be the most feasible solution to embedded products.

5.2. Priority Inversion

We created the priority inversion scenario by running three tasks at low, medium, and high priorities, with the low and high priority tasks competing for the same resource. Below is an occurrence of priority inversion (the yellow arrow) that captured from a software analyzer. The tCyclicTask, tWorkTask, and tSlowTask correspond to tasks with high, medium, and low priorities.

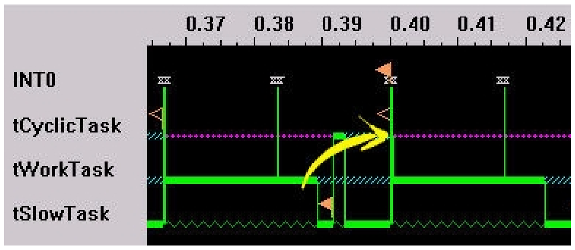

Figure 2: Measuring Priority Inversion Using Software Analyzer

We took several time measurements between tCyclicTask requesting the resource and tSlowTask releasing it, and the results are given in the table 2.

	VxWorks	RTLinux
Priority Inversion (µS) Mean (Std)	123 (1.67)	108 (0.41)

Table 2: Priority Inversion Measurments

An important characteristic of RTOSes is predictability. Although RTLinux takes less time to resolve a priority inversion problem, both figures appear to be in an acceptable range. This indicates that both RTOSes have implemented an effective priority inheritance protocol to ensure that critical deadlines are met.

5.3. Interrupt Latency

In this experiment, we configured the MPC8260 hardware timer with a period of 50 MHz to generate a timer interrupt every 20 µs. An interrupt service routine that updates a system tick count is hooked to the interrupt vector table. We use the hardware logic analyzer to measure the time between the assertion of the timer interrupt and the execution of the ISR (Figure 3).

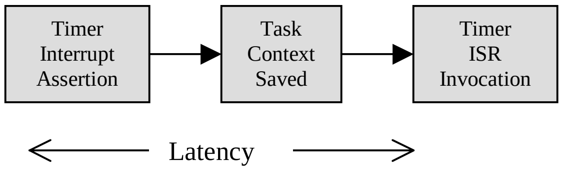

Figure 3: Interrupt Latency Test Setup

Noticed that all other system interrupts are disabled so that our measurements are not affected by nested-interrupts. The average and standard deviation of both systems interrupt latencies are recorded in Table 3.

	VxWorks	RTLinux
Interrupt Latency (µS) Mean (Std)	98 (0.55)	132 (1.2)

Table 3: Interrupt Latency Measurements

It is not surprised that VxWorks has much lower interrupt latency (35%) than RTLinux. Traditional Linux is notorious for having high interrupt latency. It appears that even though RTLinux had been added with real-time capability, it still exhibits some non real-time behaviour.

5.4. Synchronization

In our test, we only focused on measuring the time to acquire a *binary* semaphore in both systems. To measure the semaphore overhead, we first created and initialized the semaphore itself to make it *unavailable*.

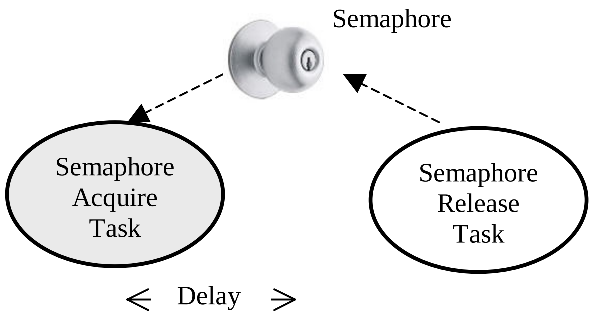

Figure 4: Binary Semaphore Test Setup

We then spawned two tasks to release and acquire the semaphore respectively, in the exact order. Finally, we measured the time (Figure 4) during which the task made the system call to acquire the semaphore. This task should not be blocked waiting since the first task should release the semaphore prior to execution of the second task. Table 4 shows the average overhead for VxWorks and RTLinux to successfully acquire a semaphore.

	VxWorks	RTLinux
Semaphore Take (µS) Mean (Std)	13 (0.29)	15 (0.08)

Table 4: Binary Semaphores Take Measurements

These figures show that the RTLinux takes slightly longer to obtain a binary semaphore than VxWorks.

5.5. Inter-Process Communication

This test is to measure the communication delay necessary for a task to send a message to another task via a message queue as shown in Figure 2.

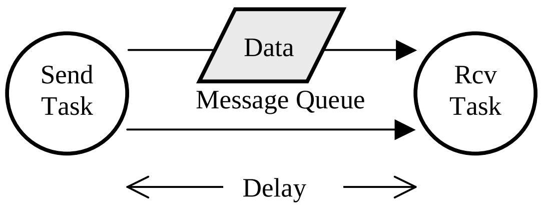

Figure 5: Message Queue Test Setup

We began this test by creating and activating (or open) a message queue. Next, we spawned a receiving task from which the message receive function is invoked. The *receive* system call blocks the receiving task and put it in the wait state (since the message queue is empty). While the receiving task was waiting for the message, we spawned a sending task to send a message via the same message queue. The time between the sending task to call the message send function and the receiving task to receive message notification is given in Table 6.

	VxWorks	RTLinux
Msg Queue Delay (µS) Mean (Std)	118 (0.9)	113 (1.8)

Table 5: Message Queue Measurements

In terms of message send/receive latency, RTLinux achieves a better score than VxWorks by a small margin. As mentioned earlier, these figures can vary greatly depending on the IPC implementation (IPC can be implemented using shared memory).

6. Conclusions and Future Work

In this project, we measured several real-time operating system key metrics to evaluate the performance of VxWorks and RTLinux. The results presented in this paper roughly matches with the characteristics of the two operating systems. Our overall analysis shows that both operating systems are suitable for real-time application. In particular, VxWorks is more deterministic and predictable, Due to time constraint and limited resources, we can focused only on studying the heart of the operating system – the kernel level performance that unveil the true system behaviour. Modern real-time operating systems often packaged with powerful run-time libraries, scalable networking components and flexible file system. A broad range of tests that cover these aspects will provide us a comprehensive result in terms of performance versus cost. Thereby, it is difficult to conclude which operating system is superior to the other without an exhaustive comparison.

7. Bibliography

[1] D. Levine, S. Flores-Gaitan, C. D. Dill, and D. C. Schmidt, "Measuring OS Support for Real-Time CORBA ORBs", in 4th IEEE International Workshop on Object-oriented Real-Time Dependable Systems 00', Santa Babara, California, Jan. 27-29.

[2] K. Obenland, "Real-Time Performance of Standards Based Commercial Operating Systems"

[3] V. Sohal, "How To Really Measure Real-Time", Embedded System Conference, Spring 2001

[4] Jun Sun, "Interrupt Latency", Monta Vista Software,http://www.mvista.com/realtime/latency/

[5] D. Stewart, "Measuring Execution Time and Real-Time Performance", Embedded System Conference, Spring 2001

[6] Real Time magazine, "Evaluation Report Definition", http://www.realtime-info.be, March 1999

[7] R. Appleton, "Understanding a Context Switch Benchmark", Linux Journal http://www2.linuxjournal.com/ljissues/issue57/2941.html, Jan. 1997

[9] V. Yodaiken, "An Introduction to Real-Time Linux", http://www.rtlinux.org/documents/RTLinux.ppt

[8] Victor Yodaiken, "The RTLunix Approach to Hard Real-Time", http://rtlinux.org/documents/papers/whitepaper.html, Oct. 1997

[9] P. Wilshire, "Installing RTLinux", http://rtlinux.org/documents/installation_june_2000.html, 2000

[10] WindRiver Systems Inc, *Tornado User's Guide*, Alameda,CA: WindRiver Systems, Inc, 1999

[11] WindRiver Systems Inc, *VxWorks Programmer's Guide*, Alameda,CA: WindRiver Systems, Inc, 1999