Measuring the Real-Time Operating System Performance

Krzysztof M. Sacha
Institute of Control and Computation Engineering
Warsaw University of Technology
00-665 Warsaw, Poland
e-mail: k.sacha@ia.pw.edu.pl

Abstract

Tests for measuring the real-time operating system performance belong to the software technology. They give quantitative measures for the most important characteristics and allow the implementer to compare the efficiency of various operating systems. This paper presents a set of simple tests for measuring the real-time operating system characteristics. The tests require no specialized hardware, and are based on direct measuring of the duration of execution of sequences of system functions. All tests described in the paper have been applied to measure the efficiency of a distributed operating system QNX. The measurements include the speed of intertask communication through messages, proxies and signals, and the speed of task switching and timer interrupt handling. The results are presented in a series of easily readable tables.

1 Introduction

Measuring operating system characteristics and comparing the systems to each other are difficult problems. The difficulty comes primarily from the fact that various systems, even if intended for the same application area, can exhibit different functionalities and no consensus exists on which features are to be measured and compared. The second source of problems is the question how these features shall be measured and how the results can be compared.

The scope of this paper is limited to the area of real-time operating systems. Such systems can be characterized by two requirements which are essential within this application area: *timeliness* and *dependability*. These requirements are implemented within the current technology using the following principles:

- multitasking,
- efficient task communication and synchronization,
- efficient time and event handling,
- transparent distribution in a local area network.

The above list gives a guideline for planning a set of measures of the real-time system performance. Unfortunately, a variety of different tools are used within various operating systems to implement those principles. A list of the most popular tools for the task communication, task synchronization, and event signaling can be summarized as follows [1]:

- synchronous messages and rendezvous,
- asynchronous messages and pipes,
- semaphores and proxies,
- signals,
- direct interrupt handling.

It is worth noting that pipes and signals have recently been standardized by IEC/ISO [2].

The goal of this paper is to suggest a set of simple tests which can be used for practical measurement of the operating system efficiency. The tests will relay on software tools only, and will not require any special hardware support. To achieve this goal, the paper is organized as follows. Tests for measuring the duration of a message transfer and the communication through a pipe is described in Section 2. Tests for measuring the speed of the task synchronization through proxies and signals is described in Section 3. A test for measuring the duration of the task switching is described in Section 4. A test for measuring the duration of the timer interrupt handling is described in Section 5.

All tests described in this paper have been implemented and applied to a distributed real-time operating system QNX. The test bed used throughout the testing process consisted of three PC-compatible computers, numbered 1, 2 and 3, and connected to a segment of an Ethernet network. The computers 1 and 2 were additionally linked to a small three-node Arcnet network. The detailed characteristics of the hardware are the following:

- Computer 1: 486/66 MHz, Opti chipset, Net-400 ECT Ethernet card (NE2000 compatible), Quantum Arcnet card.
- Computer 2: 486/66 MHz, Micronics LX30WB chipset, Net-400 ECT, Quantum Arcnet card.
- Computer 3: 486/66 MHz, UM82C480 chipset, 3Com503 Ethernet card.
- Ethernet: CSMA/CD, 10 Mbps.
- Arcnet: token passing, 2.5 Mbps.

Most of the tests were run on a single computer as well as throughout the networks. The network load during the tests was very low. This was arranged deliberately, as the intention of this paper was to measure the characteristics of the operating system itself, and not the behavior of a communication subsystem under varying load.

2 Tests for message passing

Basic tools for the task communication fall into two different categories: asynchronous, i.e. buffered, message transfer (mailboxes and pipes) and synchronous message transfer (synchronous messages and rendezvous). This diversity causes a problem, as both types of tools exhibit different characteristics and are usually applied in different context. Fortunately, there exists a specific form of the task communication, which can easily and directly be implemented using any of these tools. This is the rendezvous, implemented by a pair of message transfers: from one of the cooperating tasks to the other one and reverse. The proposal for a test procedure described below is based on a direct measuring of the time of a series of rendezvous.

Consider a single message transfer between two tasks. Although simple, this operation is not atomic and two different actions must be performed by the operating system between the beginning of *Send_message* instruction in one task and the end of *Receive_message* instruction in the other. First, the operating system must deliver the message. Second, it must switch the tasks, i.e. preempt the sending task and run the receiving one. The question arises, what exactly is to be measured: the time of delivery only, or rather the total time of the entire operation?

The answer suggested in this paper is, that the total time of the whole operation should be considered. This is the smallest piece of operation which can be observed in real life, and the division of this function into more elementary actions depends on the operating system architecture and is irrelevant for an application programmer. A test which refers to this very practical point of view can be described as follows.

Test description. Two tasks take part in the experiment. One is a client, which sends requests and the other one is a server, which responds to the client. The client task reads the current time from a real-time clock and executes a known number of rendezvous in a loop. After finishing the loop, the client reads the time again, calculates the difference and divides it by the number of loops. The result is the duration of a single rendezvous. The duration of a rendezvous divided by two equals to the time of a single message transfer.

It can easily be seen, that the test can be used for testing the speed of communication within a single computer as well as for testing the communication across the network (if the operating system supports transparent network communication).

Case study. QNX operating system supports the task communication through messages, which can be sent synchronously by means of message passing primitives *Send-Receive-&-Reply*, or asynchronously through a pipe. The duration of a single message transfer can be measured in both cases by similar test programs which kernel parts are shown in Figure 1.

The results received for synchronous message transfer are gathered in Table 1. The first glance at the table reveals, that basically identical computer hardware can be surprisingly different. Relatively small speed of long message transfer within the computer 1 suggests poor design of the cache installed on the main board. Relatively fast transfer of short messages between the computer 3 and the other ones suggests that the network card installed in this computer is faster than the other cards. Apart of these hardware-dependent effects, there are several lessons which can be learnt from the results of this test:

1. Message transfer is extremely fast within a single computer, and much, much slower in a distributed system.
2. The correspondence between the size of a message and the time needed for a single message transfer is strongly nonlinear.
3. Ethernet-based communication is extremely fast: Messages below 100 bytes can be transferred within 1 ms, and the application-level transfer rate of 900,000 bytes per second is really moving.
4. Transfer of very short messages is faster on Arcnet than on Ethernet — this effect comes probably from the existence of a lower bound on the length of frames transmitted through Ethernet.
5. Transfer of very long messages is extremely slow on Arcnet.

The results received for pipe message transfer are

Client task **(QNX messages)** **Server task**

```
cnt=count;
clock_gettime(&t1);
do {                                          do {
   Send(server,msg1,msg2,n1,n2);                 Receive(client,msg1,n1);
} while (--cnt);                                 Reply(client,msg2,n2);
clock_gettime(&t2);                           } while (1);
TM=(t2-t1)/count/2;
```

Client task **(pipes)** **Server task**

```
cnt=count;
clock_gettime(&t1);
do {                                          do {
   write(pipe1,msg1,n1);                         read(pipe1,msg1,n1);
   read(pipe2,msg2,n2);                          write(pipe2,msg2,n2);
} while (--cnt);                              } while (1);
clock_gettime(&t2);
TP=(t2-t1)/count/2;
```

Figure 1: Tests for the QNX messages and pipes.

Message length	Single computer			Ethernet			Arcnet
	1	2	3	1-2	1-3	2-3	1-2
[bytes]	[μs]	[μs]	[μs]	[μs]	[μs]	[μs]	[μs]
1	22	21	21	906	799	752	622
10	21	20	21	909	801	755	691
100	23	22	22	1074	975	934	1405
1000	48	37	38	2800	2908	2797	7191
10000	474	330	318	11512	12108	12435	52886

Table 1: The duration of a single message transfer (TM).

Message Length	Single computer			Ethernet			Arcnet
	1	2	3	1-2	1-3	2-3	1-2
[bytes]	[μs]	[μs]	[μs]	[μs]	[μs]	[μs]	[μs]
1	245	202	205	1904	1725	1622	1515
10	244	201	206	1919	1707	1613	1600
100	257	210	216	2108	1904	1809	8321
1000	368	280	283	3786	3776	3659	7974
10000	1511	1056	1225	14577	15064	14659	61333

Table 2: The duration of a pipe message transfer (TP).

gathered in Table 2. It can be noted, that pipe message transfer within a single computer is an order of magnitude slower than synchronous message transfer. This results from the implementation of QNX: Pipes are handled by a pipe manager task and each data transfer is accomplished by two rendezvous between the cooperating tasks and the pipe manager. When the network communication is considered, the time of the physical network transfer becomes a dominating factor and the results obtained for pipes and message transfer primitives, particularly for long messages, are similar. The efficiency of the Ethernet-based network communication is still very high. The most surprising figure in Table 2 is the duration of the 100 bytes long

message transfer through Arcnet. I cannot find any explanation for this peculiarity.

3 Tests for task synchronization and event handling

Tools for passing control signals between tasks can also be classified in two distinct categories: asynchronous signals, i.e. POSIX signals which look like software generated interrupts, and synchronous signals, transferred usually by means of semaphores. The semantics of these tools is different, however both of them can be used in such a way that one task will be waiting until a signal is generated by the other one. Hence, a rendezvous between two tasks can be arranged by forcing them to send a signal back and forth in a loop. The proposal for a test procedure described below is based on a direct measuring of the time of a series of "signal-rendezvous".

A semaphore is a task-level tool, and the implementation of an intertask rendezvous is obvious. POSIX signals, however, can act at a lower level of signal handlers, and two different schemes of communication are possible. The first one relates to signals transferred at the task level only, the other one addresses the relation between a task and a signal handling routine. Despite the difference, the general idea of measuring the operating system characteristics is in both cases similar and similar to the idea presented in the previous section.

Consider the signal transfer at the task level first. The actions performed by the operating system between the time instant when a signal has been raised and the time instant in which a task is starting to react on that signal include: signal generation, signal delivery, signal catching, and task switching. I believe that all these elementary actions constitute a single operation, indivisible at the task level. A test program based on this premise will be described a little bit later in this section.

Consider the signal transfer at the level of a signal handler. This case differs from the previous one in that the time of the propagation of a signal from a signal handler to the task level and the duration of the task switch should be excluded.

Test description. Two tasks take part in the experiment and synchronize with each other, sending signals back and forth in a loop. The client task sends a request-signal and waits for a response-signal from the server. The server task waits for a request-signal and sends a response-signal to the client. The duration of this loop can be measured and the time of a single "signal-rendezvous" can be calculated. A signal handling routines installed in both tasks do nothing. Nevertheless, they are necessary according to the rules of signal handling [2].

The client task reads the time from a real-time clock twice: before and after executing the loop. After finishing the loop, the client calculates the difference and divides it by the doubled number of loop cycles. This gives the duration of a single signal transfer.

The measurement of the duration of a signal transfer at the signal handler level is more difficult. However, only a slight modification to the previous test program is needed. First, the instruction for sending (raising) a response-signal must be moved from the body of the server task to its signal handler. Second, the priority of the server must be decreased, to prevent scheduling this task for execution. In this way, the activity of the server becomes limited to the body of its signal handler. The client task remains unchanged, and it performs its action exactly the same way as it did in the previous test. The result of the test is the duration of a signal transfer between the client task and the signal handler.

Both tests can be used for testing the speed of intertask communication within a single computer as well as for testing the communication between tasks executed in a distributed network.

Case study. QNX operating system supports POSIX signals, which can be sent and waited for at the task level by means of standard primitives *kill* and *sigsuspend*. The function *kill* can also be invoked from within a signal handler. The duration of a single message transfer can be measured at both levels by similar test programs, which kernel parts are shown in Figure 2.

The results received are gathered in Table 3. The lessons which can be learnt from these results are the following:

1. Signal transfer within a single computer is unexpectedly slow — it takes twice as much time as a message transfer (Table 1).
2. Task-to-handler signal transfer across the Ethernet network is fast — it takes only as much time as a single message transfer.
3. Task-to-handler signal transfer across the Arcnet network is 50% slower than a message transfer — probably a separate acknowledgement frame is sent on the network.
4. Task-to-task signal transfer across a network is surprisingly slow, particularly on Arcnet network. These results suggest the existence of hidden net-

Client task **(task level)** **Server task**

```
cnt=count;
clock_gettime(&t1);
do {                                          do {
   kill(server,SIGUSR1);                         sigsuspend(empty_mask);
   sigsuspend(empty_mask);                       kill(client,SIGUSR1);
} while (--cnt);                              } while (1);
clock_gettime(&t2);
TT=(t2-t1)/count/2;
```

(signal handler level — added to the previous program)

```
cnt=count;                                    /* SIGUSR2 signal handler */
clock_gettime(&t1);
do {                                          void handler(sig)
   kill(server,SIGUSR2);                      {
   sigsuspend(empty_mask);                       kill(client,SIGUSR2);
} while (--cnt);                              }
clock_gettime(&t2);
t=(t2-t1)/count;
TH=t-TT;
```

Figure 2: Test for the QNX signals.

Loop count	Single computer			Ethernet			Arcnet
	1	2	3	1-2	1-3	2-3	1-2
	μs	μs	μs	μs	μs	μs	μs
Task-to-handler signal transfer (TH)							
3000	42	39	39	925	855	803	974
6000	43	40	39	906	828	832	970
9000	32	39	39	902	914	793	976
Task-to-task signal transfer (TT)							
3000	51	46	45	1292	1244	1185	2147
6000	51	46	46	1292	1250	1185	2363
9000	51	46	46	1292	1247	1213	2586

Table 3: The duration of a signal transfer.

Message length	Single computer			Ethernet			Arcnet
	1	2	3	1-2	1-3	2-3	1-2
bytes	μs	μs	μs	μs	μs	μs	μs
1	26	24	24	862	762	723	600
10	26	24	25	866	760	730	606
100	28	25	25	868	765	725	604

Table 4: The duration of a proxy message transfer.

work frames, sent by the operating system in the course of a signal delivery. I cannot find any explanation why does this last so long on Arcnet.

QNX version of a semaphore is called a *proxy* and is a slight extension of the classical semaphore definition. Unlike the semaphore, which conveys only pure synchronization signals, a proxy can convey constant messages, stored within the proxy at the time of

its creation. A task can *Trigger* a proxy, or *Receive* the message stored within a proxy. The test program for the proxy communication is nearly identical to the programs shown in Figures 1 and 2.

The results received are gathered in Table 4. It can be seen, that the proxy communication is very fast in QNX, and it takes only as much time as a short message transfer. The speed of communication does not depend on the length of the message tied to a proxy, and is in any case much faster than the signal transfer.

4 A test for the speed of task switching

The time consumed by the task switching represents a pure overhead introduced by any multitasking operating system. As it was noted in the discussion in Section 2, such an overhead is inevitable in any form of intertask communication within the same network node. Therefore, the length of this time is an important feature, which characterizes the overall operating system efficiency.

A task switch occurs irregularly during the normal computer operation, and is difficult to catch and measure. Fortunately, many operating systems implement a special *Yield* function, which enforces the preemptive scheduler to switch between the ready tasks at the same level of priority. The test procedure suggested in this paper is based on a direct measuring of the time of a series of switches enforced by this function.

Test description. Two tasks take part in the experiment. Each of them goes through a loop and yields control to the partner in each loop cycle. The main task reads the current time from a real-time clock and executes a known number of the loop cycles. After finishing the loop, the task reads the time again, calculates the difference and divides it by the number of loop cycles. The result comprises the duration of a loop cycle within the main task, plus the duration of a loop cycle within the other task plus the duration of two task switches. Because the tasks are identical, a division by two gives the duration of a single loop cycle plus a task switch. The duration of an empty loop cycle can be measured separately, thus enabling the test program to calculate the time of a single task switch.

Loop count	Computer		
	1	2	3
	μs	μs	μs
10000	7.7	7.2	7.3
20000	7.7	7.2	7.4
30000	7.7	7.2	7.3

Table 5: The duration of the task switch.

In fact, the time measured in this test can include also other types of the operating system activities, e.g. periodical execution of certain system tasks. To minimize these effects, the priority of the tasks engaged in this test should be set as high as possible. If the maximal priority is set, then the activity of other tasks would be suppressed.

Case study. The test program for measuring the speed of task switching in QNX operating system is presented in Figure 3. The body of the main task consists of the reference loop, the creation of an interweaving task, and the test loop.

The results received are gathered in Table 5. The task switch appears very fast, and the predictability of the system behavior is also pretty good.

5 A test for the speed of interrupt handling

The response time of a real-time operating system depends in many cases on the speed of interrupt handling. The most important characteristic relevant to this problem is the delay between the time instant in which an interrupt has been requested, and the start of an application-oriented interrupt handler. Unfortunately, this parameter cannot be measured by means of software tools only. Therefore, the test procedure suggested in this paper attempts to evaluate another characteristic, which is the length of a period of time from an interrupt request to the ultimate return from an interrupt servicing routine. Half of this time can be considered an estimation of the delay to start an interrupt handler.

Interrupts are, in general, requested by peripheral devices irregularly, and the duration of a short interrupt servicing routine cannot be measured directly by a software tool. There exists, however, an exception to this rule — timer interrupt, which is requested regularly at a given rate. Timer frequency can usually be set under program control, and the duration of a test program execution in the presence of timer interrupts can directly be measured. If the test is run twice, with the timer set to different rates, then the duration of the timer interrupt servicing routine can easily be calculated.

Test description. The heart of a test program is a routine, which consumes a constant amount of time

Main task **Interweaving task**

```
cnt=count;
clock_gettime(&t1);
do {
} while (--cnt);
clock_gettime(&t2);
tl=(t2-t1)/count/2;
spawn(...)              /* -> task creation -> */
cnt=count;
clock_gettime(&t1);
do {                                          do {
   Yield();                                      Yield();
} while (--cnt);                              } while(1);
clock_gettime(&t2);
ty=(t2-t1)/count;
TS=ty-tl;
```

Figure 3: Test for the QNX task switch.

T (unknown). This routine is executed twice, with the timer set to different rates to give a tick every t_1 or t_2 milliseconds. The duration of each execution is measured, giving different values of T_1 and T_2 seconds, respectively. Any of these values is a sum of the time T plus the amount of time consumed by the timer interrupt servicing routine. Hence:

$$T_1 = T + \frac{T_1}{t_1}\Delta t$$

$$T_2 = T + \frac{T_2}{t_2}\Delta t$$

where Δt denotes the time of a single interrupt handling. The above equations can be resolved, giving the expression:

$$\Delta t = \frac{(T_1 - T_2)t_1 t_2}{T_1 t_2 - T_2 t_1}$$

Test case. QNX implements a system function *clock_setres* which allows to set the resolution of a real-time clock in the range from 0.5 ms to 50 ms. According to the QNX documentation [3], this function scales the hardware timer rate (interrupt 0) rather than some software algorithm. The test program has been constructed exactly as described in the previous paragraph. Its main routine is executed twice, with the clock resolution set to 1 ms and 50 ms, respectively.

A minor problem of this test consists in poor resolution of the time measurement, which is equal to the resolution of the real-time clock. To get the accuracy not worse than 5%, the test had to be conducted long enough, to yield the difference $(T_1 - T_2)$ greater than 1 second.

The results received are gathered in Table 6. They show that the timer interrupt handling takes approximately 16 μs, in any of the three computers employed during the test. Half of this value can be considered an estimation of the interrupt response time.

References

[1] W. Halang and K. Sacha, *Real-Time Systems: Implementation of Industrial Computerised Process Automation*, World Scientific, 1992.

[2] ISO/IEC 9945-1 (IEEE Std 1003.1) *Information Technology — Portable Operating System Interface (POSIX)*, 1990.

[3] *Watcom C Library Reference for QNX*, vol. 2, qnx_hint_attach function, 1993.

Test length	Computer		
	1	2	3
s	μs	μs	μs
75	15.5	16.5	15.5
75	15.5	16.5	15.5
75	15.5	16.5	15.5

Table 6: The duration of the timer interrupt servicing routine.