Time-Constrained Programming in Windows NT Environment

Leonardo Jelenković, Leo Budin

University of Zagreb

Faculty of Electrical Engineering and Computing

Unska 3, HR-10000 Zagreb, Croatia

{leonardo.jelenkovic, leo.budin}@fer.hr

Abstract **- This paper presents some experiments on Windows NT operating system trying to determine its timing limits. Experiments were concentrated on the context switching abilities in a multithreaded environment. For this purpose a program simulator is written and simulations performed. We detailed some characteristic times for different process priority classes and processor's speed. In this way we try to determine usage of this operating system in time-constrained (real-time) environment. For comparison simulations were also performed on Windows 98 and Linux operating systems.**

I. INTRODUCTION

The x86-based architecture has been making big steps into embedded-systems. Is it necessary to always build a custom program or operating system for them or can we use an existing one such as Windows™ NT? There are many reasons for using such operating system in embedded-system design. First, a tremendous amount of products and applications are available for it together with technical support from a great number of experienced users and the best development tools are available for them. Furthermore, companies are looking to standardize on only a few architectures and operating systems to keep development costs manageable. Also, users are familiar with it and those systems that look as it are popular with end users. With its graphical and networking facilities they simplify system development, freeing developers to focus on the application instead of the interfaces.

Purposes of embedded systems vary from high accurate, demanded and life critical to entertainment and fun but in almost all applications they respond on specific external stimulus. In other words such systems must have adequate real-time capabilities. Designed for a desktop environment, observed operating system lacks many of the features that bring deterministic behavior to real-time operating systems. Therefore, there already exist some third-party real-time extensions for almost every member of the Windows family, particularly for Windows NT.

In this paper we examine only current possibilities of the Windows NT alone, and also for comparison, Windows 98 and Linux, without any extensions from third party suppliers.

II. REAL-TIME PROGRAMMING FEATURES OF THE OBSERVED OPERATING SYSTEMS

We can briefly describe real-time system as system where timely responses to the external requests are vital for the system and must be satisfied. In this paper we do not observe situations with great schedule complexity where there is a big problem to find adequate schedule to meet task's deadlines. Instead, we focus on the ability of the operating system to handle events and threads with highest possible priority trying to meet their deadlines.

There are several settled requirements that an operating system must meet in order to be considered as a real-time operating system. The operating system must be multithreaded and preemptive and must support thread priority what ensures that higher priority task preempts lower priority tasks. The system has to support predictable thread synchronization mechanism and a system priority inheritance must exist. The operating system behavior must be predictable what means that information about system interrupts levels, calls and timing must be detailed.

Target operating systems fulfill some of these demands. All of them are multithreaded, preemptive, support different priority classes for threads and processes and within each class different priority levels.

Windows 98/NT have four priority classes (idle, normal, high and real-time) for processes and seven priority levels in each class (idle, below normal, normal, above normal, highest and time-critical) for threads. Base thread priority is then calculated from its process priority class and its priority level. Each thread has a dynamic priority. Threads with higher dynamic priority preempt threads with lower priority, while threads with the same priority share processor's time in a timesharing policy. The operating system can boost and lower the dynamic priority of a thread to enhance its responsiveness, i.e. when a process is brought to or remove from the foreground. The system does not boost threads within real-time processes. On NT this boosting can be controlled or even disabled. Windows NT and Windows 98 behave different when a priority inversion occurs. Priority inversion occurs is situations when a high priority thread is locked on an object that owns a low priority thread. Another thread with lower priority from first, but higher than low priority thread will get all CPU time. In Windows NT scheduler solves this problem by randomly boosting the priority of the ready

low priority threads. This ensures that with a few scheduling rounds low priority thread that owns the lock exit from critical section and high priority thread then continue its execution. In Windows 98 system recognizes low priority thread that hold a lock on which depend high priority thread. It then boosts a priority of a low priority thread up to the priority of the high priority thread.

Linux has three priority classes that have different scheduling policy. First, non real-time policy works as timeshare. Threads in this priority class are scheduled in a fair scheduling policy where all threads get some CPU time regarding to the ‘nice’ priority of their processes. Other two policies are real-time policies, one with timeshare policy and adjustable time slice (round robin) and the other with first-in-first-out (FIFO) policy. Threads in these policies are scheduled regarding its priority, thus real-time thread with higher priority preempts lower priority real-time thread. Real-time policies in Linux are available only with superuser privileges.

III. THE REAL-TIME SYSTEM MODEL

When an extern event occurs that need to be processed it invokes hardware interrupt. Regarding interrupt attributes special procedure is called. In Windows NT this procedure is named *interrupt service routine* (ISR). Its function is to determine event that pulled interrupt and to make only necessary processing. It then usually notifies the system and put an entry in a special queue. It is important that ISR finishes as quickly as possible because while it executes no other interrupt with same or low priority cannot be accepted and processed. Additional event processing will then be done according the queue at lower priority level. Because this queue is a FIFO structure one high priority task can be done after maybe many lower priority tasks whose were before in a queue. This disability reduces usage of Windows NT in hard real-time systems. Windows CE as operating system that arose from Windows NT and designed for embedded systems changed interrupt-handling procedure. As interrupt arises ISR is called which then do minimal processing. It then releases the *interrupt service thread* (IST) which is waiting for that interrupt and which will process that interrupt. Time from event arise to event processing highly depend on capability of the OS to handle threads. If there are more events to process it is important that higher priority events are processed first. This means that OS must schedule threads appropriately with its priorities. This work is concerned on the capability of OS to handle multiple threads with different priorities.

Our real-time system model consists of variable number of periodic and asynchronous tasks. Characteristic parameters of periodic task are its period time, computation time and deadline, as shown in Fig. 1.

It is assumed that release times of periodic task is zero, that is the task is immediately ready to start when it arises in the system. Asynchronous task has also computation time and deadline but it appears asynchronously with Poisson arrival distribution. Our simulator is organized as a multithreaded program. First, the main thread reads the periodic and asynchronous task parameters from a file and creates one thread for each event series. After that, every created thread is used as a event generator. According to the input data, these threads calculate time periods to next events and sleep until that time. When one of the thread wakes up, it assigns task for processing that event to one another thread with appropriate priority. These threads are then scheduled only by operating system. Asynchronous tasks are assumed to have shorter computation time, deadline and are more urgent than periodic tasks.

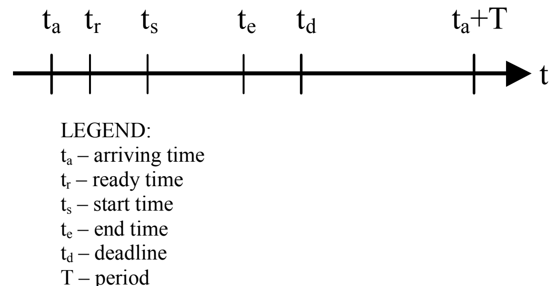

Fig. 1 Characteristic parameters of perodic task

We assume that our real-time system is running with some background load. Changing this background we can observe its influence on critical threads and their handling.

IV. EXPERIMENTAL RESULTS

Real-time model was tested on Windows NT and then for comparison also on Windows 98 and Linux operating systems on x86-based PC (Pentium™ 133MHz, Pentium MMX™ 200MHz, and Celeron™ A 300MHz). Time measurements were made using operating system built-in time functions. Background load is changed from zero-load to high load consisting of several computational extensive tasks in the background. Simulation was performed with two kinds of programs. First program always creates a new thread for every new event while second use existing thread. Events are generated as time period elapses for every event series. For this purpose *Sleep* function is used. While we can not control all system activities in simulations it happened that with two same subsequently simulations results very slightly differ. In tables were given average results for several simulations.

The shortest time period that can be achieved on Windows NT with *Sleep* function is 10 milliseconds. This time is used as period time for almost all event series involved in ours experiments. Time was measured using functions whose use a counter with frequency little above 1MHz ensuring accurate measurement in microsecond time units.

Table I show times needed for creating new thread and make it run on different processors. Threads were created within processes with different priority class: normal, high and real-time. It is given minimum time, average time and average time on loaded system achieved for thread creation.

TABLE I
THREAD CREATION TIMES ON WINDOWS NT

priority class	normal			high			real-time		
processor	t_{min} [a]	t_{ave}	t_{aveL} [b]	t_{min}	t_{ave}	t_{aveL}	t_{min}	t_{ave}	t_{aveL}
P133 [c]	714	791	830	700	836	870	700	852	880
P200	421	498	533	427	522	548	429	516	540
P300	280	356	317	292	351	345	291	335	333

[a] t_{min} and t_{ave} are minimum and average thread creation time in microseconds

[b] t_{aveL} is average thread creation time on loaded system in microseconds

[c] P133 stands for Pentium 133 MHz, P200 for Pentium 200 MHz MMX and P300 for Celeron A 300 MHz

TABLE II
THREAD CONTEXT SWITCH TIMES ON WINDOWS NT

priority class	Normal			high			real-time		
processor	t_1 [a]	t_4	t_{20}	t_1	t_4	t_{20}	t_1	t_4	t_{20}
P133	34	562	800	41	587	805	39	581	831
P200	15	511	477	18	509	515	18	514	516
P300	12	479	444	11	483	514	11	507	506

[a] t_1 stands for switch time when there is only one event series with 300 µsec processing time, t_4 for switching time when there is four event series with 150 µsec processing time and t_{20} for switching time when there is 20 event series every with 14 µsec processing time

Thread creation times are in range from around 300 µsec on a fast processor to almost 800 µsec on a slower processor. These times show us that thread creation is very time consuming operation and it is not suitable for time critical programs. Fig. 2. shows us how long it would take to process different numbers of events that appear at the same time if we use this method for event processing and processing time for one event is about 150 µsec.

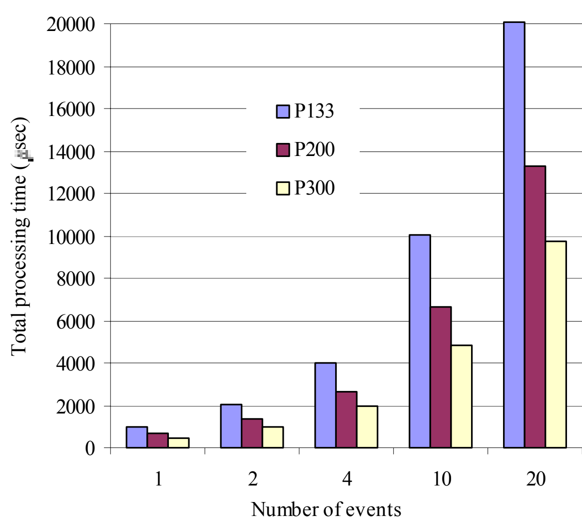

Fig. 2. Total processing time for events using method that creates a new thread for every new event (on Windows NT)

There is a very little difference in these times over different process class. High and real-time threads take a few microseconds more than threads in normal priority class.

When period elapse event generator thread release a specific thread waiting for that event. Semaphores were used as synchronization objects, so that time written in Table II includes context switching time (time needed to change current running thread with other ready thread) and semaphore operation overhead. Times under t_1 represent average context switching time between one event generator thread and a thread waiting for processing that event. This is actually average time achieved in experiments with no background load. t_4 represent maximal average switching time when there were four event series, with four threads as event generators and another four threads waiting for those events.

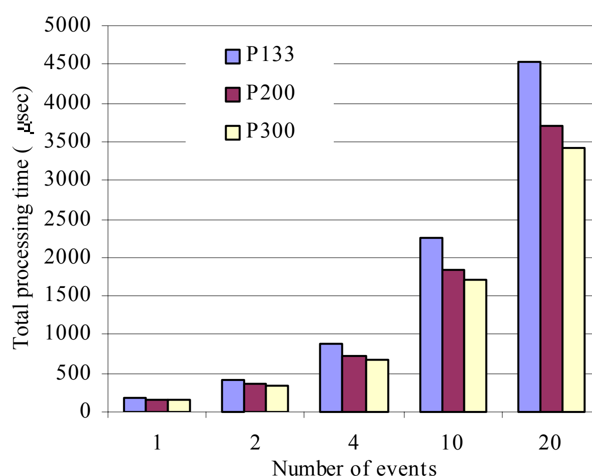

Fig. 3. Total processing time for events using method that uses existing threads for new events (on Windows NT)

As we were restricted with ten milliseconds as minimal period between two events for each series all events within series with same period arise at the same time. While event generator threads have higher priority than event processing threads (analogous to ISR and IST) they were scheduled first. First event is then processed, followed by second, third and fourth. The longest time from event arise to start of event processing in this case is time needed to schedule event generator threads increased by processing time for all but last event. This time highly depend on event processing time and for this reason processing time is set to about 150 µsec for four series and about 14 µsec for twenty series. Otherwise, total event processing time will dramatically increase over reasonable deadlines.

Form Fig. 3. we can see how long it would take to process variant number of events if we use existing threads to handle events and event-processing time is 150 µsec. Dominant factor in this method becomes event-processing time if it is significantly greater than kernel overhead from thread switching time and synchronization mechanism. This is contrary to the first method where the overhead of thread creation is even greater than useful processing.

Unfortunately, even with real-time priority class and single event series there is almost always a few situations (in ten seconds simulation time) when deadlines were not met. Table III shows average number of missed deadlines with different number of event series with different event-processing times. Reasons for these failures come from the operating system and its sometime unpredictable system activity. These, even rare, unexpected delays in response to events show that Windows NT lacks predictable behavior that an OS must have to be used in hard real-time systems.

TABLE III
NUMBER OF OMITTED DEADLINES ON WINDOWS NT[a]

#series	processing(µsec)	#events	#failures
1	300	1000	3
2	300	2000	4
4	150	4000	10
4+1[b]	100	4170	9
7+3[c]	50	3720	17
10	33	10000	18
20	13	20000	29
30	4	30000	12
40	4	40000	24

[a] on a Pentium 200 MMX processor with 1 msec deadline
[b] four periodic and one sporadic series
[c] seven periodic and three sporadic series

The same simulator program is used on Windows 98 operating system and Pentium 200 MMX processor. Although *Sleep* function on that operating system really pauses calling thread for a given number of milliseconds down to one-millisecond time units the same parameters were used as on Windows NT. Table IV shows thread creation time and context switching time. Those times are significantly greater than on Windows NT. Thread creation and context switching times are about twice longer. Oppose to Windows NT on Windows 98 were not detected irregular response delays.

TABLE IV
THREAD CREATION AND SWITCHING TIMES ON WINDOWS 98

	thread creation (µsec)		context switch (µsec)	
priority class	t_{min}	t_{ave}	t_{min}	t_{ave}
normal	776	899	35	44
high	777	908	45	52
real-time	782	805	45	50

The same simulator program was ported to Linux with necessary minor change. These changes include thread manipulation calls and time measurement functions. Although Linux posses sleep function that gets time in microsecond (*usleep*) it actually wait in much longer milliseconds time units (10-30) in this kernel version (2.2.3) on Pentium 200 MMX processor.

In Table V are listed characteristic times for thread operations. Thread creation times are little shorter than on Windows NT while thread switching times are almost twice longer than on Windows NT. Unexpected delays were not detected while using real-time priority threads.

TABLE V
THREAD CREATION AND SWITCHING TIMES ON LINUX

	thread creation (µsec)		context switch (µsec)	
thread priority	t_{min}	t_{ave}	t_{min}	t_{ave}
Normal	332	415	34	53
real-time (FIFO)	332	412	32	33

Fig. 4. presents a comparison of thread characteristic times achieved on all tested operating systems. Windows NT have shortest thread switching overhead while Linux has faster thread creation. Windows 98 takes around twice more time to perform those operations than them.

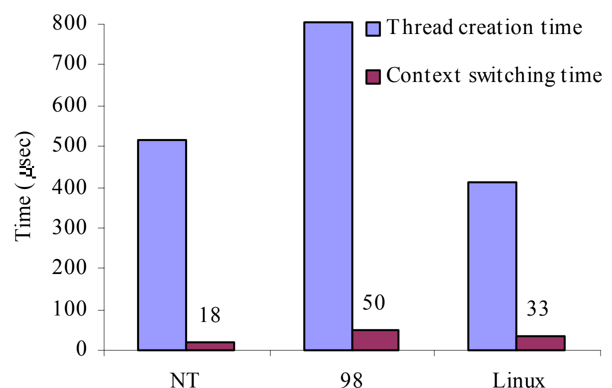

Fig. 4. Comparison of thread create time and context switching time between threads on Windows NT, Windows 98 and Linux

Forcing the system with a great number of events we can experimentally find a limit where context-switching

time between multiple threads becomes dominant and degrades system ability to react on events. If we set event processing time to minimum (few microseconds or less) number of events that system can handle in shortest period can be calculated from that period time and from context switching time. We simply divide period with doubled context switching time and get that number. Fig. 5. shows calculated and experimentally achieved maximal number of events than tested operating systems can handle in a period time of ten milliseconds on the system with same processor (Pentium 200 MMX). These numbers show us operating systems limits caused only by switching time overhead and synchronization mechanism.

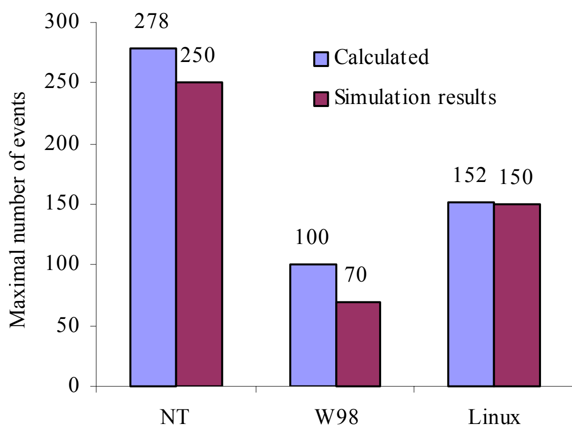

Fig. 5. Calculated and experimentally achieved maximal number of events that an OS can handle in 10 msec period using a constant number of threads for event processing

If we put some event processing time in simulator number of processed events within one period will drop down. Calculated maximal numbers and maximal numbers achieved with simulations differs significantly only on Windows 98 where overhead of context switching time grows when there are more than 70 events (140 active threads).

V. CONCLUSION

Comparing two kinds of real-time program organization, one that creates thread when an event occurs, and second that uses pre-created threads, we conclude that thread creation is very expensive operation in Windows NT. Thus, the first method is not suitable for a real-time system. With the second method we obtain acceptable results for reasonable number of event series even at very high background loads.

Increasing number of event series that explicitly increases the number of simultaneously active real-time priority threads in the system we reach the limit of about 250 events when the system can not appropriately respond due to the thread context switching time overhead. Operating systems correctly respects thread's priorities so threads with lower priority first miss their deadlines when all deadlines can not be met.

We concluded that our methodology is valuable experimental tool for timing analysis of operating systems. In our future work we are planning to include experiments with Windows CE that is claimed to be an operating system designed for embedded systems.

VI. ACKNOWLEDGMENT

This work was done within the research project "*Problem-Solving Environments in Engineering*", supported by the Ministry of Science and Technology of the Republic of Croatia.

REFERENCES

[1] N. Nissanke, *Realtime Systems*, Prentice Hall Series in Computer Science, Prentice Hall, London, 1997

[2] J. Xu and D.L. Parnas, "On Satisfying Timing Constraints in Hard-Real-Time Systems," *IEEE Trans. Software Eng.*, vol. 19, Jan. 1993, pp. 70-84.

[3] R. Gerber, S. Hong, M. Saksena, "Guaranteeing Real-Time Requirements With Resource-Based Calibration of Periodic Processes," *IEEE Trans. Software Eng.*, vol. 21, July 1995, pp. 579-592.

[4] R.A. Quinnell, "Tackle real-time applications with Windows NT," Design Feature, EDN Access, Sept. 1997

[5] A. Silberschatz and P.B. Galvin, *Operating System Concepts*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1994