Using Windows NT in Real-time Systems

Allan Baril
Spar Aerospace Ltd.
9445 Airport Road, Brampton, Ontario, Canada
abaril@spar.ca

Abstract

Due to Windows NT's widespread acceptance on desktops in business and industry, it is also being considered for use in real-time systems. The fact that Windows NT is a general-purpose operating system complicates its application into a real-time environment. We have performed an extensive performance evaluation of Windows NT to determine if we can use it in the development of future real-time systems. The evaluation involved measuring interrupt and deferred procedure call latencies as well as many other system performance aspects which would affect the performance of a real-time system. The results have provided us with insight on what types of real-time systems can and cannot be implemented with Windows NT.

1. Introduction

Windows NT is quickly becoming the de-facto standard for desktops in business and industry. As such, many people are trying to apply Windows NT as a common operating system across all uses in their businesses, which includes use in real-time systems. For example, the OMAC, Open Modular Architecture Controls, Users Group [7] is looking at Windows NT for this purpose. Windows NT is a relatively mature operating system with an extensive programming interface. There are probably more developers who are familiar with the Win32 API than those provided by other operating systems.

Of course, these strengths also lead to some of Windows NT's weaknesses when used in a real-time system. Windows NT was designed as a general-purpose operating system, with the goal to maximize visible performance to the user. To a real-time system, timely, predictable performance is much more crucial than overall system performance. An extensive performance analysis of Windows NT with respect to aspects that concern a real-time operating system was conducted to gain insight into what limitations Windows NT places on the development of a real-time system.

1.1. Overview of Windows NT Architecture

Before discussing the performance analysis conducted, it is important to have some understanding of Windows NT's internal architecture. Windows NT was originally based on a micro-kernel design, which is popular with many real-time operating systems (e.g. VxWorks, QNX). Its architecture has diverged from this though, mainly for performance reasons. For example the graphics subsystem has been incorporate into the kernel space.

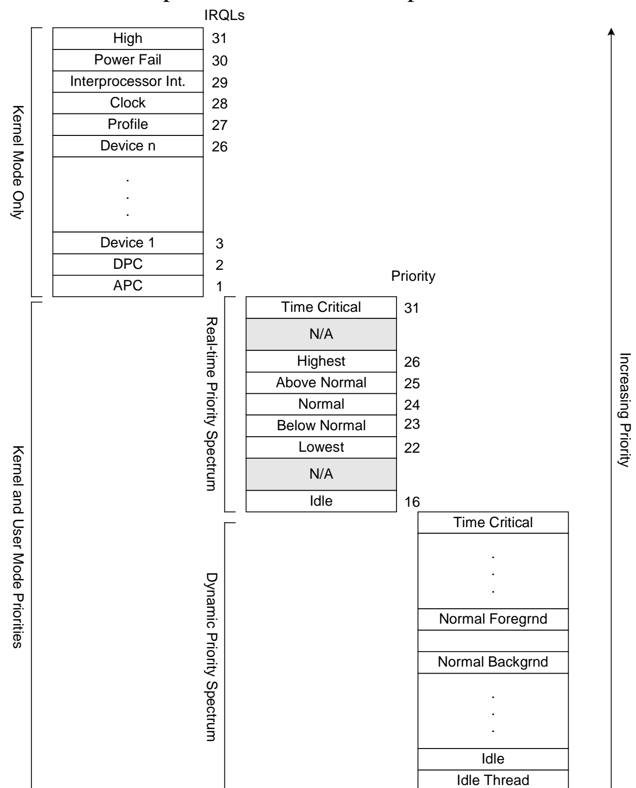


Figure 1. Priority structure

The scheduler is probably the most important piece of a real-time operating system as it is responsible for providing timely, deterministic time slices to the threads of a real-time system. Figure 1 illustrates the overall structure of the priority system provided by Windows NT. The highest priorities are given to the interrupt service routines (ISRs) for the system clock and other hardware devices. Next are the deferred procedure calls (DPCs) which are typically spawned from an ISR to continue work at a lower priority. Asynchronous procedure calls (APCs) are used to execute code in the context of a user thread. For example, the completion routine from an asynchronous file write (FileWriteEx) runs at this level. All of the priority levels discussed so far are only available to code running in kernel-mode (either the OS itself or a device driver).

The user-mode priority levels are divided into two main groups: dynamic and real-time. In certain cases, the scheduler will boost the priority of threads running in the dynamic range (for example, if it belongs to the foreground window or has returned from a driver call). This makes the system appear more responsive to the user. Unfortunately it makes it difficult, if not impossible, to provide deterministic behavior. Hence Microsoft added the real-time priority spectrum to NT. The real-time priority spectrum includes priority levels 16 to 31 (excluding 17 to 21 and 27 to 30 as these levels are unavailable). No priority boosting is conducted by the system to threads at these levels and priorities 31, 26, and 25 are higher than the system threads. As will be seen, these levels do not provide deterministic behavior.

Since NT was designed to be a general-purpose operating system, many design decisions were made to maximize overall system performance as seen by the end user. Of course some of these decisions are problematic at best to a real-time system. Windows NT uses paging to allow execution of applications which demand more memory than is physically available in the system. Paging in a real-time system can ruin all predictability. Worse, we have found that paging in one thread can affect the performance of other higher priority threads in the system. Windows NT also allows priority inversions. It attempts to solve them by boosting the priority of threads which have been starved for 3 to 4 seconds; not really a solution when dealing with events on the micro and millisecond scale.

1.2. Related Work

Interest in the topic of using Windows NT for real-time computation can be observed by the multitude of articles available in industry magazines, [3], [8], [12], from Microsoft, [5], and some previous research papers, [6], [9]. Most of this previous work either does not include actual performance values or considers only a specific type of real-time system.

Some previous performance evaluations of Windows NT have been conducted, [1], [2]. These evaluations concentrate on aspects that measure Windows NT's performance as a general-purpose operating system, instead of focusing on aspects, which affect performance of a real-time system.

2. Performance Evaluation

Microsoft is very protective of the source code for Windows NT, so at best a black box approach to performance analysis can be conducted. The performance evaluation that was conducted concerned itself with performance aspects that affect the determinism and responsiveness of a real-time system. In particular, the aspects we concerned ourselves with included: interrupt latency, interrupt execution, DPC latency, context switch time, and the effects of paging on the system.

3. Methodology

3.1. Experimental System

The testing was performed on Windows NT 4.0 with service pack 3 installed. The hardware on which the measurements were conducted consisted of a Ziatech 5080, including a Ziatech ZT5510 Single Board Computer with 166Mhz Pentium processor and 64MB of RAM. The system contained a ZT6620 Wide SCSI interface, ZT6631 Super VGA interface w/4MB running at 1024x768 with 16bit color, a ZT6610 quad serial interface, ZT5980 system utility board including a 3.2GB Hitachi DK226A-32 IDE hard drive, Sensoray 711 frame grabber, and a Seagate ZT15230WC 4.2GB SCSI hard drive.

A second machine was used to induce interrupts on the Ziatech machine by sending information through the serial ports. The second machine was a HP Vectra VL with Pentium 133Mhz, 64MB of RAM, 4 serial ports and Windows NT 4.0 with service pack 3. A HP 1650 Logic Analyzer was used to conduct some of the measurements such as the ISR latency as signals on the CompactPCI bus needed to be measured. Figure 2 provides photographs of the equipment which was used.

Figure 2. Test equipment

3.2. Applications

Several applications were developed to conduct the performance evaluations and to generate different load conditions. All of the applications were developed with Visual C++ 5.0 and the custom device drivers were developed with the Windows NT 4.0 DDK.

3.3. Load Conditions

Load condition was one of the main factors varied between different levels on most of the tests. This allowed us to evaluate the effect that different load conditions have on the performance aspect being measured. Several applications were developed and used to provide standard load conditions.

3.3.1. CPUStress

The CPUStress application is part of the Windows NT 4.0 Resource Kit. It generates a specified level of utilization on the CPU. It was used in many of the test cases to generate a constant percentage of load on the CPU.

3.3.2. Thrasher

This application was developed to produce a continuous load on the hard drive by constantly reading and writing to a file. It is used in the disk load related tests

3.3.3. NetThrasher

The NetThrasher workload actually is composed of two separate programs. One, NetThrashC, is the client while NetThrashS is the server. The client sends a block of information to the server over a network connection. The server then returns the same block back to the client. Sockets are used as the means of communication between the two systems. The NetThrasher suite was developed to produce a network load for several of the test cases.

3.3.4. DisplaySim

A workload, called DisplaySim, was designed to simulate a general application that displays information received from an interrupt-driven hardware interface. This workload uses a couple of applications. First a user application called MainDisp was run to update some text and bitmaps on the screen at a rate of 200ms. In the test system, four copies of a custom serial driver were run in conjunction with the quad serial port to simulate the interface to the hardware. Finally an application was developed for the HP Vectra to periodically send information over the different serial ports simulating the interaction with the interfaced hardware.

4. Evaluation Results

4.1. ISR Latency

Interrupt latency is very important in a real-time operating system since normally the event which needs to be handled in a timely fashion is tied to a piece of hardware which interrupts the system to let it know when it is done and/or ready to be serviced. Typically this event needs to be handled as soon as possible to ensure that the results stored in the hardware's registers are not overwritten or lost.

The test was conducted using two different methods. The first method used a custom serial device driver which handles interrupts at vector 11 from the serial port. During the driver's ISR, it reads the data register on the serial port a specified number of times and stores the results in

memory. This simulates the reading of data from the interface. At the start of the ISR, a signal on one of the pins of the parallel port is raised. A probe from the logic analyzer was attached to this parallel port pin and another probe was then attached to the PCI interrupt pin on the back of the quad serial board. By measuring the difference between the time in which the PCI interrupt was issued on the bus to the raising of the parallel port signal, we were able to calculate the latency involved from the point in time of the hardware issuing the interrupt, to the interrupt service handler in Windows NT handling it. The software on the HP Vectra would periodically send information over the serial port to cause the interrupts on the Ziatech machine.

The second method used another custom device driver called ISRTimer to measure the interrupt dispatching time of Windows NT without introducing hardware latency into the measurement. ISRTimer attaches itself to an interrupt vector, specified though the registry, and periodically issues an INT n call, where n is the interrupt vector it is attached to. This causes the processor to raise an interrupt which NT eventually routes to ISRTimer to handle. The time between raising the interrupt and its handling is measured by ISRTimer and stored internally in memory. The time stamp counter on the Pentium, as described in [4], [10], is used to get the timings which provides an accuracy relative to the processor's speed (~6.02 ns resolution for a 166Mhz processor). The timing information was then retrieved from the device driver and stored in a file so it could be analyzed.

The tests were conducted under several different load conditions. The tests were also conducted at different interrupt vectors of 3, 5, and 11, under the second method with ISRTimer, to ascertain if the level has an effect on the performance.

The following table summarizes the results using the custom serial driver:

Load	Avg. (µs)	Max. (µs)
No load	7.985	10.52
DisplaySim	12.28	21.38
25% CPU load	8.267	11.44
40% CPU load	8.095	11.6
70% CPU load	8.435	9.52
100% CPU load	8.54	9.84
40% CPU/disk load	12.46	21.16
70% CPU/disk load	12.69	30.64
Net load	11.83	40.96
40% CPU/net load	13.69	43.76
AVERAGE	10.43	21.08

As can be seen, the maximum measured overall was 43.76 µs, while the average across all the load conditions was 10.43 µs. It is important to note that the Ethernet interface raises interrupts at vector 10, so it will preempt the serial driver's ISR which was at vector 11.

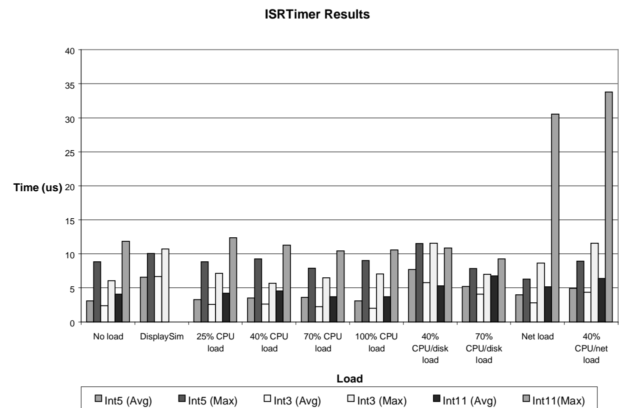


Figure 3. ISRTimer results

The results using ISRTimer, Figure 3, show that the maximum interrupt latency observed was 33.78 µs, which occurred under the 40% CPU/net load at interrupt vector 11. Without network traffic, the maximum latency was almost a third less, at 12.37 µs.

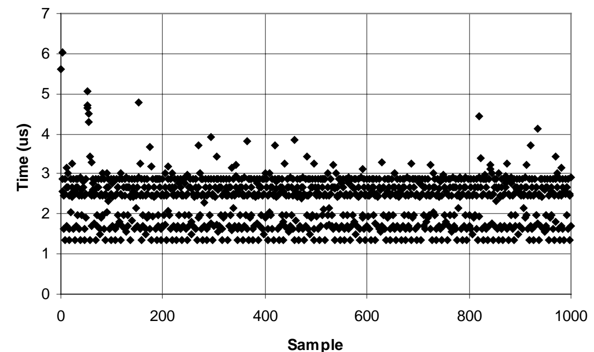


Figure 4. Interrupt latency samples

The average interrupt latency across the different interrupt vectors is a respectable 4.28 µs. The coefficients of variance were calculate and found in all cases to be quite low, ranging from a low of 0.077 for interrupt 5 with 40% CPU load to the highest value being 0.423 for interrupt 11 with 40% CPU and network load. These low coefficients of variance show that the values are fairly constant and predictable. Figure 4 illustrates the samples for the test conducted at vector 3 with no load. As can be seen, the values are pretty much grouped in the same region but there are several different layers. These layers are most likely due to interrupts, such as the system clock, preempting the interrupt of interest.

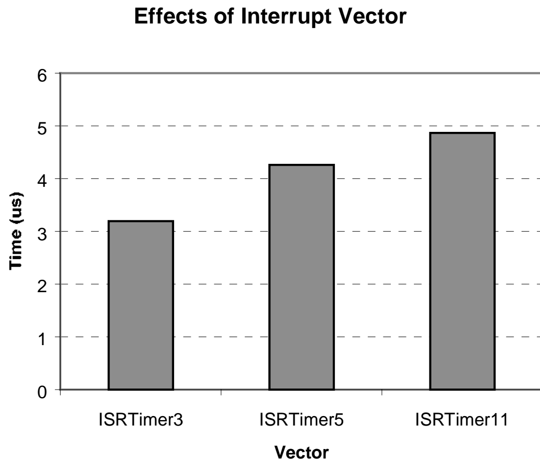


Figure 5. Effect of interrupt vectors

We can also observe the effects of different interrupt vectors. This is illustrated graphically in Figure 5. On average, there appears to be a linear relationship between the priority and the time to service. Since NT gives higher priority to lower interrupt vectors, this confirms the priority structure that NT provides to interrupt service routines.

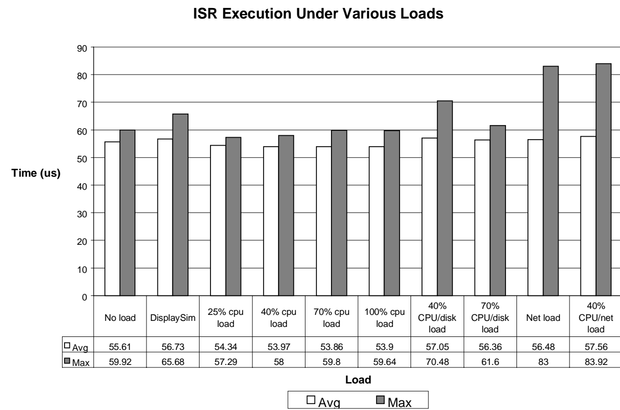
From the results, it can be observed that differing CPU loads has no noticeable effect on the latency of interrupt handling. This is as we would expect since when an interrupt occurs, assuming interrupts are enabled, the processor puts what it is doing on hold and handles the interrupt immediately. With NT's priority scheme, as long as no higher priority interrupts are currently being serviced, the interrupt handler is immediately dispatched. We do observe some delay in servicing the interrupt when load is generated which causes other interrupts, like those from the disk and Ethernet.

Finally, comparing the results from the ISRTimer values at vector 11 and those measured from the custom serial driver (which was also handling interrupts at vector 11), we can make an estimate as to the latency introduced by the hardware when handling an interrupt. Since ISRTimer generates interrupts through software, it does not have the latency due to the programmable interrupt controller (PIC) and the PCI bus. The timings for the custom serial driver are measured from the point of the interrupt being raised on the CompactPCI bus to the device's ISR being issued. The results show that the average latency introduced by the hardware is 4.22 μs. This average was calculated using only the loads where no other interrupts occur since the maximums would skew the results. It is interesting to note that the maximum times measured with the custom serial driver were observed to be greater only when other interrupts were occurring (i.e. loads with disk and network access occurring). This is presumed to be the effect of the PIC disabling other interrupts in the system until the current one is serviced.

4.2. ISR Execution

We measured the length of time that the custom serial driver's ISR executed to verify Windows NT's priority structure. Since the ISRs have the highest priority in the system, they should only be preempted by higher priority ISRs. This would indicate that the ISR execution has a relatively small number of disturbances.

The ISR execution for the custom serial driver was measured through some built-in instrumentation. At the start of the ISR, the custom serial driver would raise a signal on a parallel port pin and then lower the signal at the end of the ISR. The HP logic analyzer was then used to measure the length of time the signal was raised. The measurements were conducted under the same load conditions as for the ISR latency measurements.



	No load	DisplaySim	25% cpu load	40% cpu load	70% cpu load	100% cpu load	40% CPU/disk load	70% CPU/disk load	Net load	40% CPU/net load
Avg	55.61	56.73	54.34	53.97	53.86	53.9	57.05	56.36	56.48	57.56
Max	59.92	65.68	57.29	58	59.8	59.64	70.48	61.6	83	83.92

Figure 6. ISR execution under various loads

Figure 6 illustrates the resulting execution times measured under the different load conditions. As can be seen the average execution time is fairly stable. The maximum is fairly stable except when other interrupts are occurring. This makes sense since the custom serial driver interrupt vector is 11 and will be preempted by lower vector numbers, which includes the Ethernet driver. It is interesting to note that the disk appears to also have an effect on the ISR execution time which seems odd since the IDE controller was generating interrupts at vector 15. This could be explained by the fact that it is NT that is enforcing this priority scheme, not the hardware. So the IDE interrupt handler is immediately dispatched, preempting our driver, but as soon as the handler is dispatched, NT realizes the IDE ISR is lower priority than the currently executing one and returns back to our ISR. Unfortunately the logic analyzer used does not report standard deviations, but, as can be seen, the maximums

are fairly close to the averages which would lead us to conclude that the timings are fairly stable.

Since higher priority ISRs will preempt lower priority ISRs, we felt it would be valuable to measure the execution time for the ISRs of the drivers included with Windows NT. This gives us an idea of how much we can expect execution of an ISR to be delayed due to the execution of other ISRs in the system.

A custom device driver, Rewire, was developed to measure the execution length of different interrupt service routines in NT. Rewire replaces a targeted device driver's entry in the interrupt descriptor table (IDT) with its own interrupt handler. In its interrupt handler, it simply raises a signal level on the parallel port, sets up the stack so the targeted driver's interrupt handler will think it received the dispatch, calls the targeted interrupt handler which it replaced, and when the system returns from the call it lowers the signal level on the parallel port. Using the logic analyzer, we measured the length of the signal to determine the ISR execution time for various drivers.

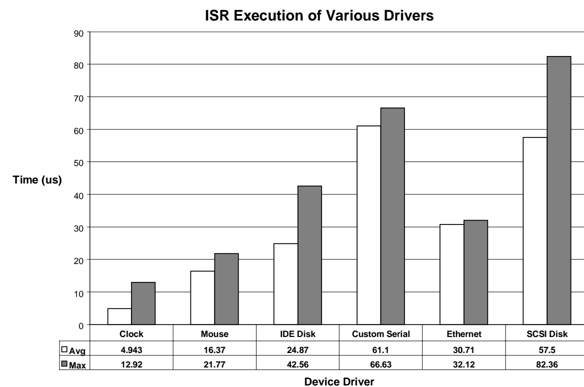


Figure 7. ISR execution of various drivers

Figure 7 illustrates the results from the measurements. Measurements were conducted for the NT system clock, the Microsoft serial mouse driver, the standard IDE (ATAPI) driver, the Adaptec AIC-78xx PCI SCSI (AIC78xx) driver, the DEC 21140 Ethernet (DC21X4) driver, and the custom serial driver. As we can see, the interrupt service routines of these drivers are relatively short. Comparing the measurements generated with the instrumentation in the custom serial driver with those generated through the use of Rewire show they are similar, but about 5 μs higher in the case of Rewire's values. This indicates that the values generated through Rewire are fairly accurate and the extra time is most likely due to some overhead generated by Rewire and/or the instrumentation in the serial driver.

4.3. DPC Latency

The next priority level after the ISRs, as seen in Figure 1, is the DPC queue. Since only a limited amount of work can be done at the ISR level, most real-time systems will probably need to conduct some tasks at the DPC level or below.

The custom serial device driver was used to measure the DPC latency. After completing the ISR, the driver issues a DPC, allowing us to measure its dispatch time. Instead of using the parallel port and logic analyzer, as we did for the ISR related timings, these measurements used the time stamp counter in the Pentium to measure the time between issuing the DPC and the start of execution of the DPC. The test was conducted several times under the same load conditions as the previous tests.

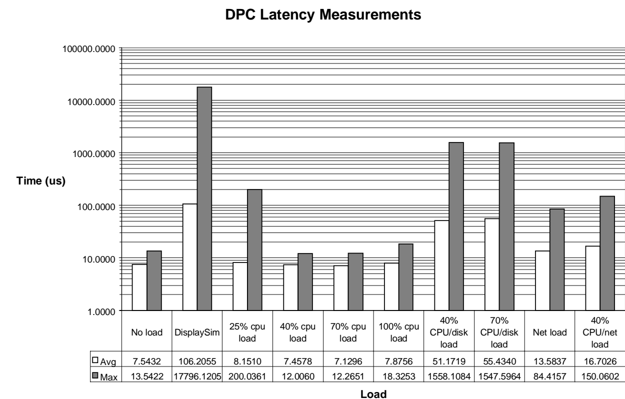


Figure 8. DPC latency measurements

Table 1. DPC latency standard deviation

Load	StDev	COV
No load	1.072	0.142
DisplaySim	1067.522	10.051
25% CPU load	6.197	0.760
40% CPU load	1.031	0.138
70% CPU load	0.962	0.135
100% CPU load	1.148	0.146
40% CPU/disk load	198.532	3.880
70% CPU/disk load	220.019	3.969
Net load	9.499	0.699
40% CPU/net load	12.414	0.743

Figure 8 and Table 1 summarize the results from the DPC queue latency measurements. These measurements confirm the fact that the DPC queue is a FIFO queue; huge delays every so often are seen due to waiting for DPCs from other interrupts to execute. For the DisplaySim load, we note an enormous delay of 17.796 ms, almost 180 times the average of 106.21 μs. The experiments later conducted regarding the effects of the page file indicate that these extreme times are most likely

due to paging activity. The disk interrupts are seen to lead to large delays of up to 1.558 ms. The DPC queue latency appears unaffected by CPU load, as it should be, since the DPC queue has higher priority then any user priority threads. The standard deviations and their coefficients of variance of the loads with other interrupts are exceptionally high. This illustrates a lack of determinism for jobs issued at the DPC level. Since all user priorities are below the DPC level, this affects the determinism of those below as well.

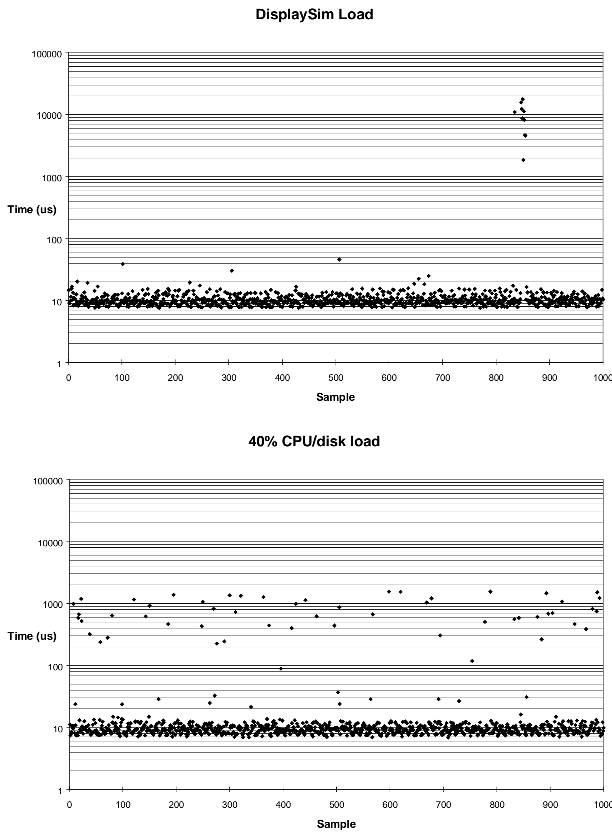


Figure 9. DPC queue latency affected by loads

Figure 9 illustrates the DPC queue latency under the DisplaySim and 40% CPU/disk load conditions. As can be seen the disturbance in the DisplaySim load only occurs once, with the remainder of the measurements being relatively predictable. In the case of the disk load, the disturbances can be seen to occur relatively frequently, which would be due to the constant read/write access of the disk.

4.4. Context Switch Latency

Context switch latency is another important system aspect often considered when comparing the performance of real-time operating systems. It is an important aspect since delays in a context switch will affect the timing and determinism of a system with multiple interacting threads.

An application called CSwitch was developed to benchmark the context switching performance of both threads and processes in Windows NT. It creates either two processes or two threads upon startup. One of the threads/processes, the client, blocks on an event, while the other, the server, signals the other thread/process. Upon signaling the other thread/process, the server then waits on another event. This causes the NT scheduler to then schedule the client since it is now ready to run. Once the client executes, it signals the event that the server is blocked on and then waits for the server to signal again. The time that the client is blocked on the event from the server is measured through the Pentium’s time stamp counter. Since each measurement actually includes two context switches, from client to server and back, the final values are divided by two. The CSwitch application upon completing a user-specified number of iterations will output the resulting times to a comma-separated file for analysis with Excel.

The CSwitch test was run under the no load, 40% CPU, and 70% CPU load conditions as used in the previous tests. It was run at priority 15 so that the load generating threads that were running at priority 12 would not interfere.

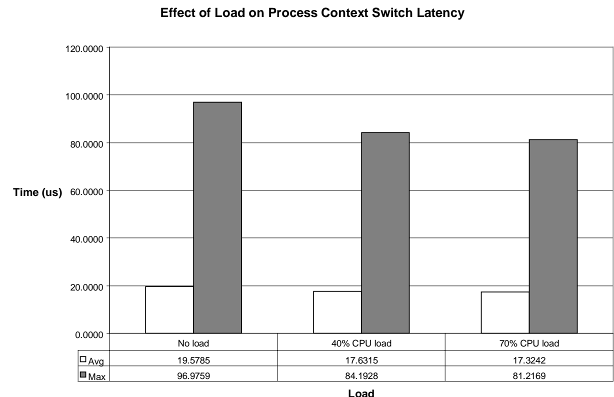


Figure 10. Effect of load on context switch latency

Figure 10 illustrates the effect of load on the context switch latency. Since the graph is relatively flat for both the maximums and averages, we can conclude that CPU load does not play a significant role on the latency of context switching.

From the measured data, we found the average context switch time between processes to be 18.18 µs, while the average context switch time between threads was 8.34 µs. A process has it’s own unique address space, while a thread shares address space with the other threads in the same process. These results confirm that NT does not

unnecessarily swap the page table when switching between threads in the same process.

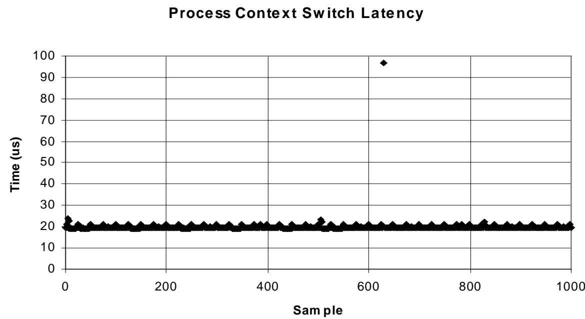


Figure 11. Process context switch latency

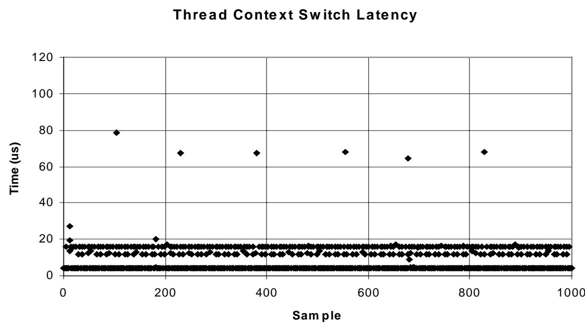


Figure 12. Thread context switch latency

Figure 11 illustrates the distribution for the process context switch latency test with no load while Figure 12 illustrates the distribution for the thread context switch latency with no load. It is interesting to note that the thread context switch has several different bands of values, while the process context switch has only one. This is most likely due to the fact that when swapping between threads in the same process, the page table does not need to be reloaded, but when swapping between threads of different processes, the page table does need to be reloaded. This would explain why the 2nd and 3rd layers are close in performance to that of the process context switch since reloading the page table is always necessary when switching processes.

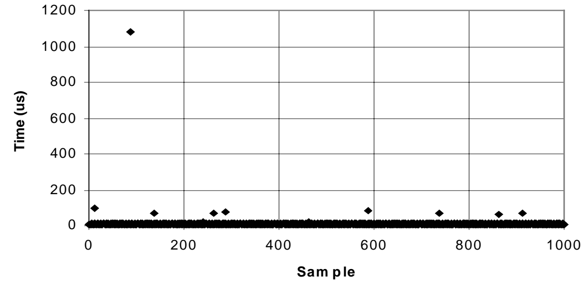


Figure 13. Thread context switch latency w/40% CPU load

Finally from the results it can be observed that there was an outlier from the average of 9.33 µs in the results measured for thread context switches under a 40% CPU load. This is illustrated in Figure 13, where it can be observed that this outlier only occurs once. Perhaps it is due to some system thread activity or some paging activity triggered by another process in the system.

4.5. Effects of Paging

Since it was felt that paging was responsible for some of the extremely large values on the DPC queue, which would then affect the determinism of all priority levels below, some extra tests were conducted to confirm whether this an accurate hypothesis. In particular the paging file was placed on several different media to see the effects. Tests were conducted as previously, with the paging file on the IDE hard drive, as well as placing the paging file on a RAM drive and SCSI drive. Since the effects of paging were suspected to be affecting the DPC latency, the DPC latency tests under the DisplaySim load were repeated, as this was the case where the anomalous results were most often seen. The tests were repeated 10 times with the page file on each media, the first 5 times leaving the DisplaySim application as is between runs, and the last 5 times minimizing and maximizing it between runs to attempt to induce extra paging.

The results generated while storing the paging file on different media are plotted in Figure 14. We can clearly see from the plot that the IDE disk device driver places DPCs on the DPC queue which can take a very long time to complete, up to 24.07 ms in one case. This is significantly longer than for both the RAM disk and the SCSI disk which had maximums of 1.40 ms and 1.05 ms respectively. This disk activity must be due to paging as there was no other disk activity being conducted during testing. This confirms the hypothesis that the large variances seen during DPC queue latency testing were due to paging activity (see Figure 9). From this data, we

can conclude that not only is it important to attempt to eliminate the paging in Windows NT to achieve determinism, but also by using a SCSI hard drive, determinism of the system can be greatly improved.

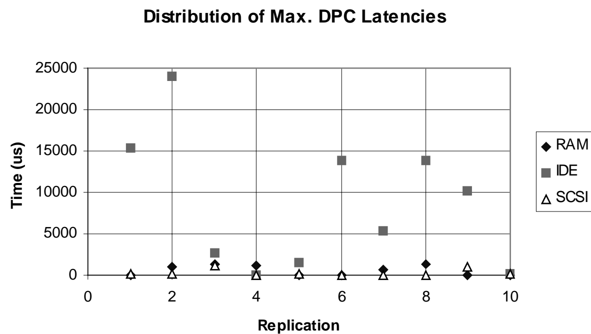


Figure 14. Distribution of max. DPC latencies

5. Conclusions

From these measurements, the interrupt service routine (ISR) latency times can be seen to be quite deterministic and predictable. If the performance of interrupt latency is measured with a particular set of hardware, the values at these levels are predictable enough to develop a hard real-time system. For example, for the system considered in the tests, one could develop a hard real-time system with 50 μs response if all the hard real-time computations were performed inside the ISRs. Maintaining all hard real-time computations in an ISR limits the types of hard real-time systems that can be developed using Windows NT. For example, ISRs cannot access the hard drive (unless it goes directly to the hardware), they cannot perform GUI code, only non-pageable memory is accessible and many of the kernel API calls cannot be used at ISR priority. The DMA support provided by the kernel API, for example, requires the calls to be made at the DPC level; thus this support could not be used in a hard real-time system using NT.

The DPC queue, the next priority level below the ISRs, was found to be quite undeterministic. Other threads in the system were found to affect the latency of the DPC queue. For example if a low priority thread issues a disk request, a DPC will eventually be placed on the DPC queue. This results in the low priority thread delaying higher priority threads and other DPCs. This clearly shows that deterministic behavior cannot be expected at the DPC queue or any priority level below, which includes all the priorities available for user applications. Only a soft real-time system, which is able to tolerate occasional delays, can be implemented at and below the DPC level. This also indicates that the effects of adding a thread to the system must be properly evaluated to ensure that higher priority tasks will still meet their required soft deadlines. This greatly complicates future growth of the system.

It was found that the DPCs created by the IDE device driver could be extremely long, up to 25ms. This, of course, leads to possibly large delays to the rest of the system. Since Windows NT implements paging, a system developed which does not use the IDE drive will still be affected by paging initiated by NT. The SCSI disk driver does not have nearly as long DPCs. SCSI DPCs were at most seen around 2ms the mark. Thus using a SCSI drive can help provide determinism in the system. A soft real-time system which is able to tolerate delays of several milliseconds can be implemented using a SCSI drive and Windows NT.

Paging, on an IDE hard drive, was found to have significant effects on system determinism. Using a package like VenturCom's Component Integrator that allows the disabling of paging is highly recommended. It is also recommended to use a SCSI hard drive if disk access is required.

5.1. Possible Improvements

This research suggests some changes to Windows NT that could vastly improve its potential for real-time computation. If Microsoft moved the priority of the DPC queue to a regular real-time priority level like 24, this would allow real-time threads attached at a higher priority such as 31 to be much more deterministic as they would no longer be affected by the delays induced from the DPC queue. Threads at this level should then be deterministic to about 100 μs.

Another possible improvement involves the removal of the DPC queue altogether. Instead of issuing DPCs, drivers could simply have real-time priority threads which remain blocked until signaled by the ISR. Upon being signaled, the thread would conduct the work which was previously done by the DPC. This would then allow the user to create threads at priorities higher than those of the drivers which would allow these threads to be significantly more deterministic. This change could theoretically be done without requiring changes to the Windows NT kernel since it only involves changes to the device drivers. Practically though it would be difficult to implement since all drivers would need to be changed or at least require the use of a controlled set of "real-time" drivers.

Finally, allowing the user to optionally disable paging in the system would greatly improve determinism of systems that do not access a hard drive. As previously mentioned, VenturCom's Component Integrator currently allows the user to do this.

6. Acknowledgements

I would like to thank all those at Spar Space Systems who have made this research possible. I would especially like to thank Dave Hiemstra for his insightful direction and Ken Martin for his support.

References

[1] J. Chen et al. The Measured Performance of Personal Computer Operating Systems. *ACM Transactions on Computer Systems*, 14:3-40, Febuary 1996.

[2] J. Chen et al. Using Latency to Evaluate Interactive System Performance. *Proceedings of the Second Symposium on Operating System Design and Implementation*, pages 185-199, October 1996.

[3] P. Cleaveland. Windows NT for real-time control: Which way to go? *I&CS Magazine*, November 1997.

[4] Intel Corporation. *Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*. Intel Corporation, 1997.

[5] Microsoft Corporation. Real-Time Systems and Microsoft Windows NT. *Microsoft Developer Network*, June 1995

[6] D. Mattern. 'Soft' Real-Time Applications Under Windows NT. *1998 International Mechanical Engineering Congress and Exposition*, November 1998.

[7] Open Modular Architecture Controls User Group. http://www.arcweb.com/omac/default.htm

[8] R. Quinnell. Tackle real-time applications with Windows NT. *EDN Magazine*, September 1997.

[9] K. Ramamritham et al. Using Windows NT for Real-Time Applications: Experimental Observations and Recommendations. *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, June 1998.

[10] M. Russinovich. Device Driver Performance Instrumenting. http://www.sysinternals.com/sysperf.htm

[11] D. Solomon. *Inside Windows NT*. Microsoft Press, 2nd edition, 1998.

[12] M. Timmerman and J. Monfret. Windows NT as Real-time OS? *Real-Time Magazine*, 2/1997