Real-Time Flight Simulators Under NT

Joseph Dube
Intergraph Corporation

Eric Anschuetz
Mark Biddle
Sam Giambarberee
Bruce Riner
NAWCTSD

Abstract

Large-scale training simulation systems have historically required a real-time operating system to function deterministically. First generation operating systems were no more than a program loader. Second generation systems, which are the bulk of the existing production simulators in existence today, have proprietary operating systems written by computer companies focused on real-time. Third generation systems, which have been in existence for about ten years, took the standard Unix kernel and modified it to have all of the real-time characteristics of the proprietary operating systems with a look and feel that was recognizable by anyone with a Unix background.

Running flight simulators under Microsoft's Windows NT would be the next evolutionary step. The main driving factors for this are the low-cost COTS hardware platforms and COTS software solutions. For Windows NT to be effective as a flight simulator operating system, it must have the ability to handle IEEE Posix.4 components such as Synchronous and Asynchronous I/O, Semaphores, Processor memory locking, shared memory, priority scheduling, fast interrupt response times and interprocess communications. Not only must NT support these functions, but also it must be modified to make these features the most time-critical functions of NT.

This paper examines the problems associated with porting flight simulation applications to NT. This will include the real-time support issues as well as GUI conversion problems associated with X-Windows to Microsoft Windows.

Author Biographies

Mr. Joseph A. Dube is a Senior Software Engineer for Intergraph Corporation in Orlando Florida. He is currently supporting Intergraph Federal Systems as the lead consultant on all Visual Simulation opportunities for Intergraph Corporation. His background includes software engineering for MultiGen, Inc., Concurrent Computer Corporation, and Harris Corporation. He has spent the last 20 years in both the simulation side and the visual side of Visual Simulation, supporting both commercial and military simulators. He holds a BSCS from Florida Atlantic University.

Mr. Eric Anschuetz is a senior Computer Engineer at the Naval Air Warfare Center Training Systems Division (NAWCTSD) in Orlando, FL, where he has worked for the past 10 years. He has worked on distributed simulations for the past 6 years, and is currently working on HLA modeling and simulation efforts in support of various related programs in the Modeling and Simulation Development Branch. He received a B.S. in both Computer Science and Mathematics from Eastern Michigan University.

Mr. Mark Biddle is a senior Computer/Electronics Engineer in the Modeling and Simulation Development Branch at the Naval Air Warfare Center Training Systems Division (NAWCTSD) in Orlando Florida. He is currently working on HLA modeling and simulation efforts in support of various related programs. He received a B.S. in electrical engineering from the Pennsylvania State University, and an M.S. in engineering management from Old Dominion University, Norfolk Virginia.

Mr. Sam T. Giambarberee is a Computer Scientist for the Naval Air Warfare Center Training Systems Division. He has several years experience in DIS networking and real-time simulations and has designed and developed the Battle Force Tactical Trainer (BFTT) voice communication and network interface software. He holds a M.S. degree in Computer Science from the University of Central Florida.

Mr. Bruce M. Riner is a Computer Scientist for the Naval Air Warfare Center Training Systems Division. He has over 15 years experience with real-time flight simulators, both in research and operational simulators, experience in replacing old host computers with newer computers systems, DIS networking and interfacing the host computer to the visual system. He holds a M.S. degree in Computer Science from the University of Central Florida.

REAL-TIME FLIGHT SIMULATORS UNDER NT

Joseph Dube, Integraph Corporation, Orlando, Florida
Eric Anschuetz, Naval Air Warfare Center TSD, Orlando, Florida
Mark Biddle, Naval Air Warfare Center TSD, Orlando, Florida
Sam Giambarberee, Naval Air Warfare Center TSD, Orlando, Florida
Bruce Riner, Naval Air Warfare Center TSD, Orlando, Florida

INTRODUCTION

The main purpose for a flight simulator is to teach a student to fly, or train a pilot in new techniques, or brush up on perishable skills. The armed forces use simulators for teaching various skills, ranging from basic flying skills to advanced high cost weapons procedures. Simulating flight usually involves three different elements: (1) the simulation of the inside of the cockpit, (2) the motion and or g-seat, and (3) the visual systems. The typical flight simulator uses a host computer system to link all the systems that build the flight simulator.

The simulation of the inside of the cockpit involves what the pilot sees, hears and does inside the cockpit. The pilot sees the various instruments, and sensors, like radar, Forward Looking InfraRed (FLIR), and weapons systems. In order to simulate the weapons systems or other complicated systems, the actual flight computers or hardware from the real airplane may be used in the simulator. The pilot also hears different sounds in the cockpit such as engines, landing gear, radios and other typical sounds. The actions of the pilot include the movement of the controls such as the joystick, instrument controls, weapons system controls, and various switches or controls.

The motion system simulates the accelerations of the aircraft; this may include a motion platform, g-seat and g-suit. However depending on the training being performed, some or all of the motion queues may not be necessary to teach the task.

The visual system is what the pilot sees outside of the airplane. Most modern flight simulators use computer graphics systems to generate the image, which can be displayed on a dome, large CRTs with special optics or other types of display devices. The computer graphics system that generates the Out The Window (OTW) image also generates the FLIR image since both are derived from the same database.

The host computer system that links the various systems, visual, motion, real aircraft flight hardware, input output system for the cockpit and aural system, is the brain of the simulator. The computer reads in the pilot's movement of the flight controls and any other switches or inputs. Then it determines the airplane's position, and any other input actions, like weapons release or lowering the landing gear. Once the computer system has computed the airplane's new position and any other status change, the computer informs the rest of the systems so the new OTW images, sounds, and motion queues can be generated.

The typical high fidelity simulator requires the position of the airplane to be computed at 60 times a second, and that data is passed to a visual system that computes a new image 60 times a second. If real aircraft flight hardware is used, which may include aircraft computers, it must be stimulated in a timely fashion in order to work properly. If the host computer system does not run in a deterministic manner, the OTW image or instruments may jump or jitter. The real flight hardware may also fail to work, if not given the proper data in a timely fashion. The pilot could even suffer from simulator sickness if the various queues are presented asynchronously to the pilot. In order to have a well-behaved simulator system, the host computer must have a deterministic or "real-time" capability.

The typical real-time computer system allows for fast interrupt response time, non-preemptive task priorities, real-time Input / Output, and memory locking to prevent swapping in a virtual memory system.

GENERIC FLIGHT SIMULATOR PORTING EXAMPLE

Over 20 years ago, NAWCTSD developed a generic flight simulator for use in such applications as carrier landings, vertical take off and landings, and general avionics training. The original code was written in FORTRAN and was developed on a Gould system running MPX in order to run in real-time. Over the years, the code was ported to other computers running various Unix flavors.

When NAWCTSD began investigating Distributed Interactive Simulation (DIS) for fast moving aircraft, the generic flight simulator became an obvious choice for an initial test-bed. Modern compilers allowed for a C language DIS front end to be interfaced with the FORTRAN code of the actual flight simulator. Initially, this code was ported to a Motorola computer system consisting of three single board computers (MVME-187), all running VMEexec, Motorola's real-time operating system. Later, this code was ported to Silicon Graphics, Hewlett Packard, and DEC Alpha computer platforms all running their various flavors of Unix.

When the High Level Architecture (HLA) was introduced to the simulation community two years ago, NAWCTSD launched an Office of Naval Research (ONR) sponsored effort to investigate the various uses of HLA. Again, the generic flight simulator was chosen as the initial HLA test-bed. Because the Runtime Infrastructure (RTI), central to HLA processing, was being distributed in Windows NT, as well as the fact that PC platforms had grown exponentially more powerful in the past 10 years and have become a viable simulation platform, it was decided to port the Unix-based flight simulator software to Windows NT.

The first task involved porting the FORTRAN code to a more modern language. A search of the web resulted in finding a "freeware" conversion program developed by AT&T called f2c that would convert a FORTRAN program to C. Normally, conversion programs result in code that has meaningless variable names, is without structure, and is not very readable. While this was somewhat true with using f2c, it was decided to go ahead with the conversion for the following reasons:

1) The original FORTRAN code looked "messy" anyway, and the conversion with f2c didn't make it any more so.

2) Having all of the code in one language makes it far more portable (e.g. not everybody has a FORTRAN compiler anymore and it is sometimes difficult to link FORTRAN code with C code).

3) We were using Microsoft Visual C++, and having all of the code in C++ made it possible to use the debugger to debug all of the code.

4) The f2c program did a very good job of converting code. Even with multiple EQUIVALENCE statements and some tricky array indexing, the code compiled and ran the first time, after the conversion.

After using f2c to convert the code to C, it was a simple matter to add formal function prototypes to make it C++ compliant. In the end, the flight model code converted to C++ was not pretty, and certainly didn't have any object-oriented design to it, but it converted well and was now able to become the core of a new HLA object oriented test-bed. The code now only needed to be ported to Windows NT from Unix.

There were several specific instances of code where modification would need to be done to get from Unix to Windows NT. These were

1) Timing code - both frame timers and general time of day functions had to be rewritten.

2) Socket interface - luckily WINSOCK's interface is almost identical to the ethernet socket interface of Unix.

3) Shared memory - the Unix flight simulator relied heavily on shared memory to communicate with the DIS core. In HLA, this section was to be rewritten with object oriented classes that are linked directly into the code and would eliminate the need for shared memory.

4) Printf and cout statements - a generic write-to-display-window routine was developed for use in Windows NT.

5) Joystick routines - a Flybox serial port joystick interface that was used on Unix platforms was rewritten for Windows NT since the serial interface was quite different. It was also supplemented with the ability to use a $20 joystick that can be plugged directly into the joystick port (read using DirectInput).

6) Sound Effects - a custom VME sound card costing $6000 to provide sound effects on the Motorola system was replaced with a $30 sound card that was driven using DirectSound extensions by Microsoft.

7) Semaphores - replaced with Windows NT equivalent critical sections.

8) Byte Swapping - although not related directly to Windows NT, Intel processors used by PCs use little-endian byte ordering as opposed to the big-endian byte ordering specified in DIS. For this reason, all data must be byte swapped upon receipt and prior to sending out on the network.

In striving to maintain compatibility with Unix systems, all of the OS dependent code was placed in separate routines that could be conditionally compiled resulting in one baseline that supported both Unix and Windows NT.

After all of the translation was complete, the generic flight simulator core was linked with NAWCTSD's Simulation Middleware Object Classes (SMOC) that serves as both an HLA and DIS interface layer. The only question was whether or not the system could sustain frame rates of at least 30 Hz. In practice, it could, but obviously frame rates were somewhat variable because Windows NT is not a real-time operating system. This was fine for a test-bed of a generic flight simulator, but may not be reasonable for an actual flight simulator.

Complementing the F14 flight model is a separate program that displays all of the cockpit instruments for either an F14 or F18 aircraft. The generic flight simulator sends packets of information to the "glass cockpit" display 30 times a second over the ethernet. The packet contains all of the information that is needed for the program to display all of the instruments with the correct data, including altitude, fuel levels, heading, and both target and aircraft time, speed, and position (TSP) information among other data. The cockpit program then uses this information to display all of the aircraft's instruments.

This cockpit program also had a rather long history and was written long before the flight simulator was converted to either DIS or HLA. Unlike the flight simulator, this program was originally written in C, on a DEC MicroVax II computer running VMS and using a DEC graphics library. Later, the program was converted to use X-Windows, which made it far more portable. The program was originally written to only display an F18 cockpit. The ability to display an F14 cockpit was added at the same time that the flight model was converted to use DIS. The program was then ported to Hewlett Packard, Silicon Graphics, DEC Alpha, and Motorola computer systems all running some flavor of Unix. A year later, the cockpit display was ported to a standard PC computer running LINUX.

By the time that the generic flight simulator was converted to Windows NT, PCs had grown powerful enough that it was possible for one PC to run both the flight simulator and the cockpit display, so it was decided to also port the display software to Windows NT. Luckily, the design of the original software was such that the low level graphics routines were confined to basically one file. Several higher level routines were written to perform such tasks as drawing rectangles, drawing circles, changing pen colors, drawing text, and so forth. These routines all called low level X Window routines to actually perform the drawing functions. It was a relatively simple matter to change merely the body of the higher level graphics functions to use the Windows 32 API (Win32) graphics calls instead of calling X Windows routines. None of the routines that actually drew the instruments needed changing because they all used the high level routines.

In addition to changing the graphics calls from X Windows to Win32, several other changes to the code were made including:

1) Timer code was changed for generating software interrupts at 30 Hz from Unix timer routines to Win32 time function calls.

2) Routines to read mouse positions were changed from X Windows calls to Win32 message handlers.

3) The program was rewritten from one that statically sized the display at 1280x1024 to a resizable window. Scaling of line drawing routines was easy, whereas fonts had to be dynamically generated in order to resize them correctly.

4) Again, byte swapping was added in order to read data coming from big-endian computers.

5) The ethernet interface had to be changed from Unix sockets to Winsock.

6) No shared memory or semaphores where used in the original Unix program, so no equivalents had to be added for the Windows NT version.

In comparing this list of changes that needed to be made for the cockpit program with the list for converting the actual simulator, it is obvious that there is a lot of overlap. In fact, graphics, timers, sockets, shared memory, and semaphores are probably some of the first areas to look at when porting any program from another operating system to Windows NT.

In the end, we were able to port a Unix-based flight simulator that ran on multiple CPUs to one that ran on a single PC under Windows NT. Sustained average frame rates of 30 Hz were achieved which was fine for our generic flight simulator. Due to the nature of Windows NT, there were a lot of small variations on the amount of time spent for any given frame. This did not have any real detrimental effect on our test-bed, but would certainly be something to beware of when real-time performance is critical to a simulation.

REQUIRED REAL-TIME FEATURES

There are many ways to define a real-time system. According to the internet news group comp.realtime.news, the definition is given as: “A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation, but also upon the time at which the result is produced.” Simply stated, a real-time operating system must, without fail, provide a response to an event within a specified time window. This response must be predictable and independent of other activities performed by the operating system on behalf of other tasks. Providing this response implies that system calls will have a specific, measured latency period. Using this definition, Windows NT is not a real-time operating system. Windows NT is a general-purpose operating system that has the capability to provide very fast response times, but not deterministic response times.

In designing a real-time application, there are several required features that the operating system must provide. These features include real-time control of I/O, interprocess communication techniques such as shared memory and semaphores, the ability to lock critical sections of code in memory, and the ability to implement priority scheduling for controlling different real-time critical tasks. The operating system must also provide for deterministic responses to interrupts.

Real time flight simulators may utilize multiple threads or processes for a variety of reasons. One purpose might be to distribute portions of the simulator on different physical processors within the same machine, such as high fidelity real-time graphics on one processor and computationally intensive search and sort algorithms on another. Another use might be to interoperate with other simulators or distributed portions of the same simulator over a network using data send/receive threads.

Shared memory is handled in NT by creating a file-mapping object. Two or more processes may open the same block of memory as though it were a file, read from it, and write to it. A process utilizes a mapped file object by invoking the CreateFile(), CreateFileMapping() and MapViewOfFile() routines. The concept of a file-mapping object is illustrated in figure 1.

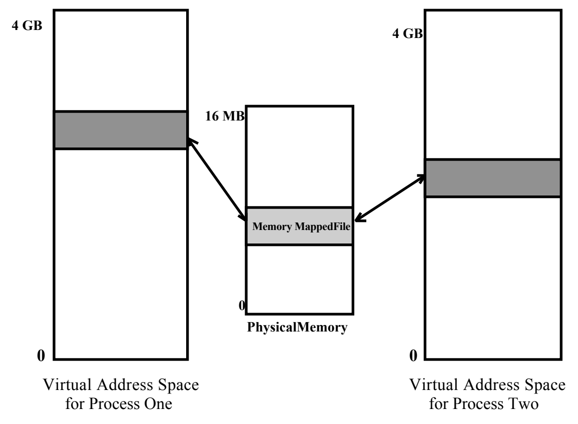
4 GB
0
Virtual Address Space for Process One
16 MB
Memory MappedFile
0
PhysicalMemory
4 GB
0
Virtual Address Space for Process Two

FIGURE 1
Creating a Shared Memory Section

Windows NT is a multitasking operating system that must handle multiple processes and threads. The term *process* refers to an executable program loaded into memory complete with all the resources assigned to that process. This includes its virtual address space, privilege mask, resource quotas, and dynamically assigned objects such as open files or shared memory blocks. Every process has at least one *thread* of execution, that is, one path that the processor follows through its code. All threads belonging to a process share the resources and assets of that process. They all follow instructions from the same code image, refer to the same global variables, write to the same private address space, and have access to the same objects. A single process can choose to divide its work into several simultaneous tasks by creating a thread for each of these tasks. When a single process runs several threads at once, it is called *multithreading.* When a single processor runs several processes at once, it is called *multitasking.* Threads should usually be considered anytime a program is involved with asynchronous activities. Threads are created quickly and they interact with each other easily, whereas creating a new process is more time consuming and involves the system loading and activating a new executable image from disk.

In terms of efficiency and speed, a single process multi-threaded design is generally preferable to a multi-process design. Process creation, switching, and destruction, for example, are slow compared to the same operations for threads. Threads also have the advantage of sharing memory and other resources, such as file descriptors or handles to graphic devices, within a process, while preserving the capability for maintaining thread-specific resources. There is less overhead required in inter-thread communication verses inter-process communication. For these reasons, the focus of this discussion is on threads as opposed to processes.

In Windows NT, threads can have one of six states, which include ready, standby, running, waiting, transition, or terminated. In ready state, a thread is available to be scheduled for execution, but has not yet been scheduled. In standby state, the thread has been scheduled and is waiting its turn to run on a processor, which happens at the next context switch. When in the running state, the thread is executing instructions on the processor until its allotted time slice runs out. In the waiting state, the thread is suspended pending some particular event. In the transitioning state, the thread is suspended pending retrieval of a required resource by the operating system. And in the terminated state, the thread ceases execution.

Windows NT uses a preemptive multitasking thread scheduling policy. Preemption of a thread can be affected by its priority, which can be set either manually through software or automatically by the operating system.

Every process has a priority rating, and threads derive their base scheduling priority from the parent process. When the system scheduler preempts one thread and looks for the next thread to run, it gives preference to threads of high priority. Once the scheduler selects the next highest priority thread to execute, it saves the context of the currently executing thread and loads the context of the new thread into the processor registers. The newly loaded thread runs for one time slice, which is likely to be 10 to 20 milliseconds[1].

NT provides several types of synchronization objects, which include critical sections, mutexes, semaphores, and events. Processes, threads, and file objects can themselves also be used as synchronization objects.

The critical section is probably the most efficient of the synchronization objects. It is similar to a mutex, but whereas a mutex can be used between threads of different processes, a critical section can only be used between threads of the same process. The state of a critical section object is either signaled or not-signaled. Programs use the critical section by requesting ownership of it. The state of the critical sections is set to not-signaled when it is in use by a thread, and any additional threads requesting ownership are put in the wait state pending release of the critical section. Once the thread with ownership of the critical section is done with it, the critical section is released, and its state is reset to signaled, making it available to other threads.

The term *critical section*, in addition to being the name of a particular synchronization object, is also used to refer in general to any section of code that is protected by a synchronization object, whatever type of object it may be.

The mutex object gets its name from the words "mutual exclusion", because access to the critical section (protected code) within a mutex is available only to the thread with ownership of the mutex. The mutex operates like a critical section. A thread which owns a mutex or critical section object may repeatedly wait for the same object without deadlocking itself, but must release the object after each wait [3].

A semaphore object allows a specified number of threads to simultaneously access a critical section. This number of threads is specified in a count, which is specified during the semaphore creation. If a thread requests use of a semaphore, and the count is greater than 1, then the thread is granted use and the count is decreased by one. Once the thread is done using the semaphore, it releases the semaphore and the count is again increased by one. If the count reaches 0, then any additional threads requesting use of the semaphore are placed in the wait state until the count is increased again, at which time the threads are granted use according to the order in which their requests were made.

Event synchronization objects are used to notify threads when a desired condition or set of conditions occurs. A thread can wait for one or more events to simultaneously reach the signaled state before continuing. After being put in the signaled state, an event needs to be reset to the not-signaled state. Events can be defined to either reset automatically, or to wait to be reset manually through software.

Threads or processes can themselves be used as synchronization objects. The handle of the thread or process goes into the signaled state when the thread or process terminates. Finally, a file may be used as a synchronization object. Its handle goes into the signaled state upon completion of file I/O operations.

In the Scalable Architecture for Distributed Interactive Systems[2] project, single processor pentiums were used as a low cost approach to handling thousands of entities being generated in a distributed interactive training environment. The primary concern was to give highest priority to the thread processing incoming network packets such that no incoming packets would be dropped or lost. Setting the priority for this thread using the system routine SetThreadPriority() did not change the fact that NT assigned a minimum amount of time to handle other computable threads when they obtained the processor. These time slices ranged from 10ms to 20ms in a rather unpredictable manner. During the time that other threads had the processor it was observed that the thread handling the network I/O was dropping several packets. The results suggested that more packets were arriving during these time slices than the buffers could accommodate. At an incoming rate of 1000 packets/second, at least 10 packets were dropped for each time slice devoted to non-T1 threads (one packet per millisecond). At an incoming rate of 2000 packets/second, the dropped packet rate increased to 20 packets per timeslice.

This was unacceptable and indicated that NT threads were not suitable for this type of real-time process. In the end it was necessary to utilize a single process design that did nonblocking network reads by periodically polling the receive sockets during idle frame time. It was also necessary to perform occasional I/O polling during compute-intensive sections of code.

The use of Windows NT in this project precluded the precise timing implementation demanded by real-time simulations. Frame timing was implemented using NT's TimeGetTime() routine which returns the time since NT was started. Resolution was set at 1 millisecond via the TimeBeginPeriod() routine. These functions were used in a polling fashion by the PC acting as the simulator host to provide frame syncs to a network of PCs as shown in figure 2.

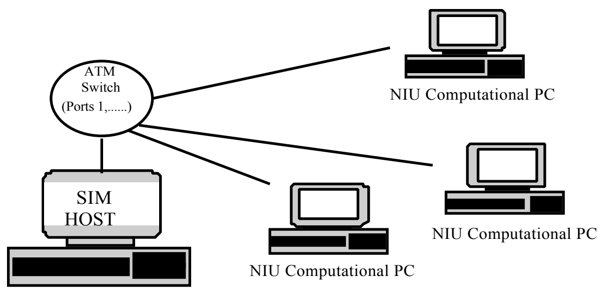

FIGURE 2
Scalable Architecture

Windows NT does not provide a technique for handling interrupt driven I/O events, and this affected synchronization among the networked PCs (each PC received the frame syncs at different times). To use Windows NT effectively in this application, a better frame synchronization method would have to be implemented.

SURVEY OF REAL-TIME MICROSOFT NT COTS SOLUTIONS

Since Microsoft NT is at least a portion of the synchronization problem, let's explore ways to decrease its variability when responding to interrupts.

Deterministic Response Times

Trying to decide which Real Time OS (RTOS) to use in the Flight Simulation domain can be a daunting task. As it turns out, there are over 40 RTOS's now on the market trying to solve one or another flavor of the real time problem. This group of 40 RTOS's can be divided into groups if you consider each group as a function of the time an interrupt occurs requesting a process to run, until that process runs (Figure 3.)

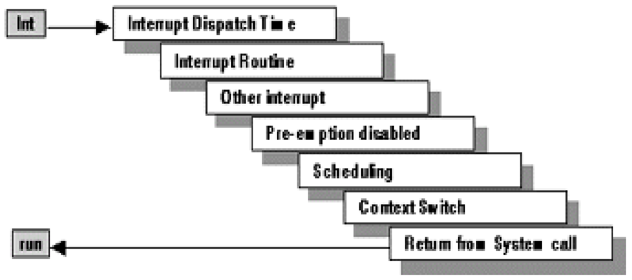

FIGURE 3

Dividing up the RTOS's into three groups: (1)Embedded OS's, (2)RTOS's running with Microsoft NT and (3)Microsoft NT modified for Real Time. Examples of these groups are Wind River's VXWorks for Embedded OS's, VentureCom's RTX for an RTOS and Windows CE for a modified Windows. All three of these Operating Systems have development environments that run under standard Microsoft NT, so from a user's perspective, they are similar. Each of these three groups can be delineated by how fast they perform the steps described in figure 1 - the time from receipt of an interrupt through the executing of the first instruction of the interrupting process.

So, given these three categories, where do flight simulators fit? Working simulators of today are driven with computers designed for worst-case process dispatch latencies of 200 microseconds (Harris NightHawks and SGI with REACT/pro). If we use this number as the determining factor in choosing a Microsoft NT environment, we can see from Table 1 that RTOS's running on the same processor as Microsoft NT will solve the flight simulation deterministic response time problem.

Deterministic Response Times

	nsec	1-100 usec	> 5 msec
Embedded OS (VxWorks)	X		
RTOS w/NT (VentureCom)		X	
NT modified for Real Time (Windows CE)		X	
Standard NT			X

Table 1

In the cases where VentureCom was used, it should be pointed out that RTX 4.1 provides a Real Time Application Programming Interface, known as RTAPI 2.0. If a process uses RTAPI-only, then it can achieve worst-case response times on a properly configured system of less then 50 microseconds. However if a process uses other Windows NT API's in combination with RTAPI, then the process can only achieve "soft" worst-case response times of around 1 to 5 milliseconds. Both methods and their respective response times may be simultaneously employed on the same system - that is, some processes use RTAPI-only, other processes use RTAPI with other API's, and the remaining processes use just the Windows NT API's without RTAPI.

Windows CE will have similar constraints, but for the most part solves this problem by eliminating a bunch of Win32 API functions (500 out of 1000) that seem to have been the system bottlenecks. On the plus side, Windows CE runs in a lot less memory space which is great for embedded applications like hand-held computers where memory is a scarce resource. On the minus side, if you currently have a Windows NT/95 application, it may not compile and work will be required to use only the Windows CE API calls.

Multiprocessor Systems

Operating Systems such as Wind River's VxWorks supports Multiprocessing today in a fashion similar to distributed systems connected together via a high-speed reflective memory structure. This type of architecture has worked in the past when one physical CPU was not powerful enough to drive all the components of the simulator, and is still being used today.

Multiprocessor functionality is just becoming available under RTOS's such as RTX. VentureCom has a beta version of RTX where one CPU is running Microsoft NT with the other CPU running their real time Kernel, RTX. When the Deschutes chip arrives this fall, which supports up to eight processors, RTX should be able to do the deterministic real time processing across multiple processors that is required for the high-end 6DOF and Weapons Systems Trainers that are today fielded on proprietary hardware and Operating Systems.

Windows CE is a preemptively scheduled, multithreaded operating system, just like Windows 95, Windows 98, and every version of Windows NT. One difference is in the maximum number of processes that the operating system can support. On the desktop, the limit is system memory. On Windows CE, the maximum number of processes at any one moment is 32. On a Handheld PC, the shell and bundled applications create 10 processes, leaving room for 22 user processes. The limit on the number of threads in the system is much higher and is only limited by available system memory. On Windows CE, as on desktop versions of Windows, threads are the unit of scheduling and every thread has a stack and a priority. Every process that starts running gets one thread, and from there processes can create more threads. If you want to port an application to Windows CE that uses a large number of processes on another OS, one way around the maximum process limit involves using threads. Combine two or more processes into one process, with each sub-process having its own thread. Obviously, this requires having a reasonable way to combine processes. Work will start to pile up if, for example, two processes to be combined had a lot of global variables with the same names, or if the same function names were used.

Conclusions

Comparing the three Operating Systems (out of the 40 or so available), some comparisons can be drawn.

1. Of the three, Wind River's VxWorks has been around the longest and therefore is more mature. It supports a more Unix-like interface but is rapidly moving over to support those customers who prefer the Microsoft development environment.

2. Windows CE is the least mature. There are planned enhancements such as nested interrupts, better thread response, additional task priorities, and semaphores which will allow immediate response to external events and interrupts, which will not be released until second quarter, 1999. Windows CE seems to have been initially designed for hand held computers (i.e. palm computers). By the first half of next year, Microsoft has promised to enhance Windows CE to support demanding real time and mission critical applications.

3. VentureCom's RTX develops and runs under standard Microsoft NT and if one is familiar with that development environment, they would feel right at home. RTX is more like the existing flight simulator operating systems where the development and run-time environments are the same. VentureCom's SMP support is a must for flight simulators and should be available at this year's I/ITSEC. They additionally plan on supporting Windows CE, which will allow a single development environment for RTX on both embedded and non-embedded computers.

All three approaches have one common weakness – device drivers. This is not a new problem to the real-time community. When a device interrupts and takes control of the system, the worst-case interrupt response time becomes bounded by the behavior of the driver. While the BIOS and Microsoft Windows NT kernel operations that mask interrupts keep real-time operations to a worst-case interrupt response time of 20-50 microseconds, device drivers may be unbounded. Currently the only solution is to test devices for well-behaved drivers.

Real Time under NT, even with some of the extensions described in this paper, may not quite be the guaranteed response time solution yet. But the key word is yet. Fifteen years ago the same arguments were used when Unix solutions started appearing in the flight simulation arena. Today Unix kernels that have been modified for real time are operating some of the most demanding flight simulators built in the last fifteen years. Microsoft NT with real time extensions is poised as the next real time OS.

REFERENCES

[1] Mastering Windows NT Programming, 1993, Brian Myers, Eric Hamer

[2] A Scalable Architecture For Distributed Interactive Systems.
1997 Interservice/Industry Training, Simulation and Education Conference.
William Rowan, Sam Giambarberee

[3] Multithreaded Programming with Windows NT,
1996 Prentice Hall, Inc.
Thuan Q.Pham and Pankaj K. Garg

BIBLIOGRAPHY

Microsoft Corporation, April 6, 1998. Microsoft Announces Plans to Expand Arena for Embedded Applications With Windows CE by Addition of Hard Real-Time Capabilities.

Microsoft Corporation, April 8, 1998. Microsoft to Incorporate VetureCom's Component Technology Into Future Versions of Windows CE.

Microsoft Corporation. FAQ for Windows CE Version 2.1

Microsoft Corporation. Real-Time with Windows NT.

Timmerman Jr., Martin "Windows NT as Real-Time OS?" Realtime Magazine, Q2, 1997.

Quinnell, Richard A. "Tackle real-time applications with Windows NT" EDN Magazine, Sept. 12, 1997.