A Comparison of OS/390 and Windows NT

MVS SCP Project
SHARE 92, Session 2828
San Francisco, California
22 February 1999

James Antognini

antognini @ us.ibm.com

IBM T. J. Watson Research Center
P. O. Box 704
Yorktown Heights NY 10598-0704

Abstract

"He ain't heavy. He's my brother." OS/390 and WinNT? OK, would you believe second cousins? The two systems are more alike than you may think. You've probably heard that WinNT has address spaces. But how about page faults? Memory-mapped files? Did you know that in WinNT a user-mode application invokes kernel-mode routines by "SVC calls"? That scheduling employs an execution-state hierarchy that recalls MVS's lock hierarchy? Or that a device driver consists of a front-end to start I/O, an interrupt-driven back-end and a scheduled I/O-completion routine? (Can you spell I-O-S D-R-I-V-E-R, S-L-I-H and S-R-B?) These and other similarities, and a few dissimilarities, of OS/390 and WinNT in architecture and internals will be considered in pondering the two systems as a case of convergent evolution.

Formatted on 24 September 1999 at 15:05:34

Change History

This paper is an expansion of a predecessor paper in the SHARE 91 (Summer 1998) *Proceedings*. Changes from the version published in the SHARE 92 (Winter 1999) *Proceedings* are marked.

17 March 1999 —

- OS/390 has time-slicing in the dispatcher.

24 September 1999 —

- Masking of interrupts by higher IRQLs isn't necessarily done in hardware.

Acknowledgement

The author gratefully acknowledges information and comments from Mark Russinovich of IBM T. J. Watson Research Center, Greg Dyck, Stephen J. Kinder and Peter Relson of IBM OS/390 Development, and W. Anthony Mason of Open Systems Resources (OSR). Any mistakes are of course the author's responsibility.

Trademarks

The following are registered trademarks or trademarks of the International Business Machines Corporation: IBM, MVS, MVS/ESA, MVS/XA, OS/390 and RACF.

Microsoft, Windows, Windows NT and Windows 2000 are registered trademarks or trademarks of the Microsoft Corporation.

Intel and Pentium are registered trademarks of the Intel Corporation.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Table of Contents

Introduction

For a long time Microsoft's Windows was regarded merely as a desktop system, the home of single-user applications like Word for Windows, Quicken, Internet browsers, E-mail and so forth. With Windows NT (which originally meant "New Technology" but which, like "DVD," has come to mean nothing more than itself), Windows has grown up. Introduced to general use in the second half of 1993, NT has gone through several major revisions. It promises to challenge Unix in that system's various avatars as a multi-user operating system and as a server operating system.[1] With NT's newest version likely to appear this year or next, the OS seems to be anointed for ever-bigger things.

Windows NT does not squarely confront OS/390 in the latter's role as a multiuser, server and, especially, transaction-support operating system, but the OS/390 community is most definitely conscious of NT, if only because applications on NT are commonly used as front-ends (terminal clients, FTP clients and browsers) to OS/390 back-ends (TSO, Unix System Services, FTP, CICS and DB2). But although the OS/390 community knows about Windows NT and many use it on a daily basis, understanding of NT is commonly confined to end applications that run on it. And the knowledge is often accompanied by pre-conceived notions, if not prejudice. "It's a desktop system." "It's the younger, evil brother that laid OS/2 low." It's just … "Not There"?

In fact Windows NT has more history than many realize. A great effort went into it, and it has grown from OS/2's younger sibling to an adult weighing in at 30 to 40 million lines of code in Windows 2000 (né NT 5).[2] 30-40 MLOC (to use developer parlance) probably puts NT in the same order-of-magnitude ballpark as OS/390.

This paper will compare and contrast Windows NT with OS/390, looking for similarities and the occasional striking difference. The reader is assumed to be well-versed in OS/390 design, development and use, so there will be little explanation of OS/390 operations.

With one or two exceptions, it is Windows NT 4.0, Service Pack 4, that will be the basis of examining NT, as that is the latest version in widespread use. Further, it is the x86-based implementation that will be discussed, because the reader is thought likely to be using that platform and because the author's NT experience is on that platform. For OS/390, version 2.6 will be the standard.

---

[1] The reader interested in a comparison of Unix and Windows NT should consult Mark Russinovich, "Windows NT and VMS: The Rest of the Story" and "NT vs. UNIX: Is One Substantially Better?," *Windows NT Magazine*, December 1998.

[2] 30 or so million, according to Giga Information group, *Computerworld* Online News, 06/09/98 05:34 PM; http://www.computerworld.com/ home/ news.nsf/all/9806092nt. 35 million, according to Gartner Group, http://www.zdnet.com/pcmag/insites/ dvorak/jd981116.htm. 30-40 million, according to Steve Ballmer (although it wasn't clear if he was referring to NT 4 or Windows 2000), http://www.microsoft.com/ presspass/exec/steve/9-28bizapps.htm.

The Ascent of Windows NT

In the minds of most people, CP/M begat DOS, which begat OS/2, whose younger brother Windows NT is. The strands of influence are a little more complicated than that.

Microsoft did develop DOS on the base of Digital Research's CP/M, and Microsoft and IBM created OS/2 in partnership. Then Microsoft developed early versions of Windows (e.g., Windows 386) and realized phenomenal success in Windows 3.0. Finally, Microsoft birthed Windows NT as version 3.0.

That, however, is but part of NT's evolutionary tree. David Cutler, NT's chief architect, had worked at Digital Equipment for more than 15 years and brought VMS into being as the OS of DEC's popular VAX line.[3] Cutler and others started a project to create Prism (hardware) and Mica (OS), but the project was cancelled. Since the project was located in Seattle, it was perhaps only natural that Microsoft should be interested in the pool of talented DEC engineers. In 1988 Cutler and others moved to what then might have appeared as the junior (albeit already very successful) partner in IBM's plans for the PC.[4] At the same time that Microsoft and IBM were developing OS/2, the new group embarked on creating another operating system.

It remains a matter of debate how much or little OS/2 influenced Windows NT's development. To this day NT includes an OS/2 subsystem, which is to say, APIs with OS/2 personality. Whatever the IBM contribution to NT via OS/2, that would not have been the first time IBM had influence. When the company retrenched in VM development in the 1970s, some individuals left IBM's Burlington, Massachusetts, development center for near-by Digital Equipment.

There are still other strands of influence on NT. A great deal of the original GUI work (for example, icons) was done at Xerox PARC. Apple picked up that work, and did nothing much with it for quite a long time. That and related techniques appeared in Windows 386 and, in a more polished form, in Windows 3.0. Various features prominent in NT—virtual storage and preemptive multitasking come to mind—had been around in a number of systems since the 1960s and 1970s. Objects and their paraphernalia had become common currency in some circles by the early 1980s. The MACH kernel work from Carnegie Mellon University was reflected in the division of the NT core into Hardware Abstraction Layer (HAL), Kernel, Executive and subsystems; each successive layer is higher-level and bigger than the previous and is available to neighboring layers via APIs.[5]

In sum, Windows NT is based on streams of influence hardly newer than those embodied in MVT-SVS-MVS-MVS/XA-MVS/ESA-OS/390.

---

[3] NT mechanisms like multistage I/O, scheduling by software interrupts and kernel primitives were introduced at DEC in RSX-11M, an OS project Cutler headed before VMS. Personal communication, Mark Russinovich, IBM T. J. Watson Research Center.

[4] *Showstopper!*, G. Pascal Zachary, pp. 20-21.

[5] *Windows NT File System Internals*, Rajeev Nagar, p. 4.

Similarities and Differences In Structure

This section and the next treat similarities and differences between OS/390 and Windows NT. Similarities will be mixed with differences, and sometimes both will be treated under the same item. This section concentrates on similarities and differences in the systems' structure. The next section deals with a few topics that affect the two systems from a developer's perspective. An evaluation of the two operating systems will be essayed in the final section of the paper.

Address spaces

Address spaces serve several purposes: Isolation of programs from one another and from damage to each other; providing more virtual storage than is available in real memory; the medium in which most programs run (all, that is, that run with virtual addressability); a point of ownership of some resources.

In NT, an address space has 4G of virtual storage; in OS/390, 2G. In a typical NT system, the first 2G (addresses 0-7FFFFFFF) are private, and the next 2G are common, or system[6]; x86 hardware, however, allows only kernel-mode programs to touch addresses higher than 7FFFFFFF.[7] In OS/390, an address space runs from 0-7FFFFFFF. For reasons of history, common areas—CSA, SQA, LPA and nucleus—straddle the 16M line (FFFFFF), with private areas above and below those areas.[8]

In NT, *process* comprises not only address space (separately addressable virtual storage) but also threads (see below) and attributes like priority class and resources. In OS/390, *address space* includes separate virtual storage, tasks, resources and attributes like non-swappability. In both systems, an address space may be swapped out to free up storage frames, and pages may be stolen for the same purpose.

(NT does not have the OS/390 concept of a dataspace that is mapped virtually but in which no program may run with that space as the primary address space.)

Storage

In NT, storage (RAM) is composed of virtual pages, which on x86 platforms are 4K in size.[9] A page can be backed by a real frame and by paging storage on hard disk. *Paged* virtual storage may not be currently backed; *non-paged* is always backed. When page faulting cannot be tolerated, backing can be guaranteed by locking down the pages. Otherwise, paged storage is subject to stealing of the backing frames. Stolen frames are not necessarily used immediately but rather are first put on a queue from which they can be reclaimed with content intact. Page tables themselves consume enough real storage that the tables are allowed to be paged out. OS/390 has very close analogs to all these features.

An NT page may be readable and writeable. For user-mode programs, it may be readable only or neither readable nor writeable. As noted above, storage above 2G is not readable by user-mode programs. There is also a type of storage duplicated upon first write, which is handy in accommodating writeable static in DLLs and forking by programs that use NT's POSIX subsystem.[10] OS/390 storage can be readable, writeable, copy-on-write or none of these.

As noted already, both systems engage in address space swapping and page stealing to ensure an adequate reservoir of backing storage. NT seems more aggressive in stealing frames than is OS/390, stealing pages even in the absence of obvious demand.

---

[6] On WinNT Server systems, it is possible to configure the first 3G as private.

[7] *Intel Architecture Software Developer's Manual, volume 3: System Programming Guide*, p. 4-29. *Inside Windows NT*, David A. Solomon, table 5-12, p. 259.

[8] Except for the PSA, which S/390 requires to start at address 0.

[9] There is also a 4M page, used for part of the NT kernel.

[10] *Windows NT File System Internals*, p. 26.

One thing not present in NT is storage protection by key. Storage key is one place where OS/390's ancestry is evident, since earlier systems like MVT ran several programs in a single address space (as we now know it). Programs were not isolated by addressability but were subject to control of access by proper storage key. From time to time in MVS, storage keys have been employed to protect programs running in a single address space. VSPC with key 2/key 8 and, much more recently, CICS with key 8/key 9 (subsystem storage protection override) are cases in point.

Preemptive, interrupt-driven scheduling

In NT and in OS/390, a piece of work has a priority, whose characteristic is that the work is subject to losing the CPU in preemption by higher-priority work when that becomes ready. Preemption may happen when an external event such as I/O and timer interrupt takes place; if the interrupt makes some other thread ready and the latter is of higher priority, the interrupted thread loses the CPU until such time as there is no other higher-priority work.

In NT, preemption is supplemented by allocating a fixed "quantum" of time to a thread when it gets control on the CPU. When the quantum expires, the thread may be displaced from the CPU, even by other threads with the same priority (this can be done by placing the thread at the end of that priority queue or by demoting the thread in priority).

OS/390 uses "time slicing" to a similar end: After running a dynamically adjusted amount of time, an SRB or TCB is subject to losing the CPU to higher-priority ready work when the work's becoming ready was occasioned by an interruption on another CPU (if the interruption occurred on the same CPU, the SRB or TCB was of course preempted immediately). But OS/390 employs more techniques than mere time slicing to achieve something like fair scheduling.

In sum, NT and OS/390 follow a fundamentally preemptive, interrupt-driven strategy, but each supplements that in its own way. It isn't possible to make a blanket statement about the relative effectiveness of the two systems' scheduling mechanisms, and one must presume that each system has adopted the approach best suited to workloads typical of it.

Privileged mode

Windows NT features two fundamental privilege states: User mode and kernel mode, which correspond to "ring 3" and "ring 0," respectively, in x386 hardware.[11] In OS/390, problem state is not privileged, and supervisor state is. In both systems privileged state confers, amongst other things, the use of all instructions and access to resources denied to the non-privileged state (e.g., in NT, access to all 4G of an address space).

Access to kernel mode in NT can be given in several ways.

1. The boot sequence.
2. Invocation by an interrupt such as I/O or timer, by an exception (e.g., invalid opcode) or by the INT instruction.[12]
3. Receiving control from another program itself running in kernel mode. An ISR (see below in "Multistage I/O") is an example of receiving control in kernel mode from a routine that itself got control in kernel mode due to an I/O interrupt.

[11] More specifically, bits 0-1 in the CS segment register denote the Current Privilege Level, ranging from 0 to 3. Only 0 and 3 are employed in NT.

[12] The term "trap" is applied to some exceptions, for example, those arising from INT instructions. In NT, "trap handlers" service interrupts, exceptions and invocations by INT.

Except for the INT instruction, these mechanisms are recognizable from OS/390. Yet even INT will be familiar, since the instruction behaves much like OS/390's SVC instruction: Status is saved in a predefined place, control passes to a handler defined at boot time by the operating system, and the handler gives control to a routine designated by the operand of the instruction.

A very frequently used INT instruction is INT x'2E' (decimal 46), which transfers control to an NT exception handler, which parses parameters and gives control to the indicated NT routine, for example, NtCreateFile.

Leaving aside INT 3 (used to call a debugger), all INT instructions invoke routines defined by the operating system. These routines can, however, be made to invoke user-supplied device-driver routines, as shall be shown.

(It is interesting that NT does not offer something like OS/390's PC routines, which are routines given control in an authorized state[13] directly by hardware upon issuance of a PC (Program Call) instruction. Invocation of kernel-mode routines is made possible only through the INT instruction, despite the fact that x86 architecture defines call gates, which are like PC entry tables in that they are software-created specifications of entry points for privileged-execution routines.[14])

Kernel-mode code via a device driver

The only supported mechanism in NT for adding kernel-mode code is the device driver.[15] To the OS/390 developer, this may sound like requiring an IOS driver or an EXCP appendage to perform authorized functions, but in reality the process is simply founded on NT's I/O software structure, is not especially complicated and demands little knowledge of NT's I/O support, unless one is in fact writing a driver to effect, or affect, I/O.

The mechanics for setting up and using a device driver for kernel-mode functions merit examination in depth. The steps for setting up a device driver are these:

1. The registry (see below) is updated to indicate that the driver is to be loaded by NT at boot time. Such an update of course demands that the updater have authority to do it. Alternatively, at some time after booting one may invoke a program to install the driver. The functions required by such a program (such as OpenSCManager) themselves require that the program execute with proper authority. Sufficient and typically used authority with either method is Administrator,[16] the highest authority of an NT user.
2. After being loaded, the driver code is invoked for initialization at the mandatory entry point DriverEntry. The code resides in system space and executes in kernel mode.
3. DriverEntry creates a device object. If the driver isn't concerned with real I/O devices, this object is merely a logical entity, but a necessary one, as shall be seen. It will often be convenient for the device object to include an extension, since that can be the anchor for the driver's objects, structures, control blocks or what you will. The extension, like the driver code and the device object, resides in non-paged storage. The device object has a name of the driver's choosing. For example, if the driver is called Ctrl2cap, it might choose the name \Device\Ctrl2cap. (\Device denotes a place with special meaning in NT's directory-like object namespace.)

[13] Ordinarily. It doesn't make much sense to define a PC routine that would get control in a non-authorized state, albeit that is possible.

[14] *Intel Architecture Software Developer's Manual, volume 3: System Programming Guide*, pp. 4-15 through 4-19.

[15] *Inside Windows NT*, p. 65.

[16] The strictly required authority is "load and unload device drivers," which is ordinarily given only to Administrators.

4. DriverEntry creates a symbolic link to the device object name. Typically the link would take the form \DosDevices\Ctrl2cap, because \DosDevices has special meaning in object namespace (see below).

5. DriverEntry defines entry points for operations it expects to be requested.[17] In the present instance, those would be Create, DeviceControl and Close. The entry points may have any names desirable.

6. DriverEntry returns to its caller.

When a user-mode (or kernel-mode) program wishes to invoke the initialized driver, it follows several steps:

1. The program opens the device object by calling NT's CreateFile with the name \\.\Ctrl2cap specified. An NT stub routine gets control and, after relatively few instructions, issues INT x'2E'. The NT Open effector understands the file name to refer to \DosDevices\Ctrl2cap and thus to driver Ctrl2cap. Presuming the driver did not impose an authority requirement when it created the symbolic link, the effector looks up the entry point defined by the driver for a Create operation.

2. The driver receives control at its Create entry point, with PASSIVE_LEVEL IRQL (a software-defined privilege attribute, described in "IRQL" below). If the driver is satisfied that the invoker of CreateFile is a legitimate user of the driver's functions, the driver sets up any necessary structures to describe the connection the user is asking for.

3. The user-mode program gets back a handle from the successful CreateFile. When the program wants a driver function, it supplies that handle along with driver-defined parameters in calling NT's DeviceIoControl.

4. The NT DeviceIoControl effector determines what is the target of the request, copies the program's parameters from user space to non-paged storage in system space[18] and, with PASSIVE_LEVEL IRQL, invokes the driver's entry point for DeviceControl operations, passing a pointer to the copied parameters.

5. The driver examines the parameters, satisfies itself the request is proper, does whatever is necessary, and returns.

   (Note that if the copied parameters point to user-space areas and if the driver has raised its IRQL to DISPATCH_LEVEL—for example, to ensure it does not lose the CPU to another thread—, the driver may not touch the user-space areas without ensuring no page fault will occur. The areas could be locked to ensure they are backed, or the driver might queue up an Asynchronous Procedure Call (APC, described below under "Multistage I/O") to the calling thread and wait for the APC to perform its duties and signal the driver that it is done.)

6. When the program no longer needs the driver's functions, it should close the file associated with the driver. Closing will cause the driver entry point for Close to get control (again, at PASSIVE_LEVEL IRQL) so that the driver may do clean up, release of resources and whatnot.

The reader will appreciate that the driver mechanism is very similar to an SVC routine in OS/390: An SVC can be loaded at IPL or dynamically. It is invoked by the SVC instruction, which passes control to the SVC Interrupt Handler, which, after possibly checking for the caller's APF status, saves caller status (e.g., registers, PSW) and passes control to the designated SVC routine. The SVC runs in PSW key 0, supervisor state, so it has access to most system services. It is the SVC's responsibility to ensure that the request is legitimate as the code's designers intended. If the SVC gets the CPU lock or otherwise becomes

[17] In the Win32 Driver Model (WDM) of Windows 2000, DriverEntry defines only AddDevice. AddDevice, called when a device is added (at boot or dynamically, via PnP), defines the other needed entry points.

[18] The copying is predicated on the program having indicated buffered operations. If not, there is no copying, and the driver is pointed to the program-supplied parameters, which would typically be in paged (that is, pageable) storage.

disabled for external interrupts, it must ensure that it does not touch pageable storage whilst executing disabled.

Multistage I/O

The handling of I/O is perhaps the most striking similarity between Windows NT and OS/390, for in each case I/O is effected in stages, with analogous if not identical mechanisms employed in the several stages.

In NT, I/O works like this:

1. An application like Quicken opens a file. Ultimately control passes to a kernel-mode routine, which verifies the correctness and legitimacy of the request, builds necessary structures, including a file object, and passes back a handle to refer to the object.
2. Quicken asks for some I/O to the file, specifying the handle along with other particulars. Eventually control reaches a kernel-mode routine, which determines which device driver supports the I/O.
3. The device driver receives control, parses user-supplied parameters and, if the device is not currently busy, calls lower-level components that issue a physical request (on x86, an IN, OUT or related instruction) to the associated device.
4. In time the device completes the requested work and generates an I/O interrupt.
5. Due to the interrupt, NT's trap handler gets control in a random address space and calls the interrupt dispatcher. The latter determines what Interrupt Service Routine (ISR) the device-driver front-end specified, and invokes that routine.
6. The ISR is running at device IRQL, which severely restricts execution (no page faulting, for example), so the ISR does only what has to be done here (e.g., clearing the interrupt, obtaining any error status, perhaps starting another I/O operation to the device), queues up a Deferred Procedure Call (DPC) routine and returns.
7. The DPC is eventually given control (usually on the same CPU as the ISR that scheduled it). It, too, runs at a high privilege level, so it does what it can (perhaps updating buffers and control structures). The DPC, again like the ISR, is running in a random address space, so if the address space with Quicken is needed, the DPC queues up an Asynchronous Procedure Call (APC) routine and returns.
8. Eventually the APC runs under the intended thread in its address space, at a fairly low privilege level but still in kernel mode in system space. The APC can touch user buffers (that is, can page-fault) and make final status available to the Quicken caller. If the I/O request has been entirely effected, the APC marks it complete and returns.
9. Quicken recognizes the I/O request is complete.

Anyone familiar with the I/O mechanism of OS/390 from its earliest MVS days will see similarities: Access-method request, SVC, STARTIO in IOS, the SSCH instruction, interrupt, I/O FLIH gathering status and SLIH perhaps driving more I/O, SRB routine and notification of the originating program. NT and OS/390 have arrived at similar solutions to maximizing thread throughput and efficient device use in an operating system of address spaces and preemptive scheduling.

Filter drivers

One thing in NT I/O is strikingly different from anything in OS/390: The ability to intercept requests. By defining a "filter driver" in front of a file system's driver, the requests made to a file system, network

card, display or whatnot can be inspected and manipulated. For example, a filter driver can be inserted[19] above the NTFS driver, so that requests (open, read, write and the like) can be seen and perhaps changed. One purpose of a filter driver might be to encrypt/decrypt or compress/decompress a file's data. Another use could be to learn of any changes made to the file data. A third purpose is to run a virus scanner as part of file operations. In effect, an application (with proper authority, to be sure) can have an exit ("callback," in NT terms) added to the I/O path.

Interrupt Request Level (IRQL)

IRQL is a major NT mechanism and one with no direct analog in OS/390. Every executing routine has an IRQL. The system scheduler, when it gets control from a routine that voluntarily or involuntarily gave up control of its CPU, assumes the IRQL of the surrendering routine. One of the scheduler's purposes is to drain queues of work requests at the various IRQLs, so the scheduler starts looking for work that is ready to go, that can run on that CPU (some work has CPU affinity) and that requires the current IRQL. If such work exists, it is given control. Eventually there remain no suitable work requests at this IRQL, so the scheduler reduces the IRQL and starts looking at that level.

A work request is presented by means of an interrupt. In this context, "interrupt" is broadly defined to include both hardware events (for example, I/O, timer and clock interrupts) and software events (queued-up requests for routines to run). Every interrupt is associated with a specific IRQL and will not be registered (hardware event) or carried out (software request) on a CPU whilst something is running there at that IRQL. IRQL on the CPU must drop beneath interrupt IRQL for the interrupt to get service.

A kernel-mode routine may increase or decrease its IRQL. The NT functions effecting IRQL changes actually do so by pairing higher IRQLs with masking of interrupts such as I/O and timer.[20] A routine is prohibited by NT from calling functions not supporting its current IRQL (the Microsoft DDK specifies with what IRQLs a function is compatible). In effect, higher IRQL confers greater importance (less chance of losing the CPU) and more restriction (fewer functions implicitly or explicitly available).

IRQL is *not*, however, a serialization mechanism if there is more than a single CPU, for IRQL governs scheduling (including interruption masking) on the current CPU only. Although its hierarchy gives IRQL a resemblance to OS/390 locking, IRQL is not truly like OS/390 locking, for an OS/390 lock (the CPU lock excepted) serializes all CPUs in a machine image. To affect all CPUs in an NT system, a spin lock is needed (see "Serialization and synchronization" below).

The table below gives IRQLs from lowest to highest for the x86 platform.[21]

[19] Insertion is accomplished by the filter driver creating its own device object and attaching that object to the device object of the target driver (the one to be filtered). Now NT will give control first to the driver of the device object most recently attached to the target driver, and so the filter driver will see all requests made to the target.

[20] The masking isn't necessarily done in hardware. NT may allow interrupts but will not allow an associated ISR to run. Since the interrupt cannot be "seen" by anything except core code, it has been masked logically.

[21] *Inside Windows NT*, pp. 84ff. *Windows NT File System Internals*, pp. 10-11. MSDN DDK *passim.*

Name	Remarks
PASSIVE_LEVEL	"Normal" work.
APC_LEVEL	Highest level allowing thread scheduling. Used for APC routines.
DISPATCH_LEVEL	Suspends thread scheduling. Page-faulting disallowed. Used for device-driver Startio routines, for DPC routines[22] and for most spin locks.
DIRQL 1 … DIRQL *n*	Device level, e.g., an ISR (device-driver back-end). Used for ISR spin locks. I/O and timer interrupts masked. NT in effect associates a particular DIRQL with a device.
PROFILE_LEVEL	Timer used for profiling.
CLOCK1_LEVEL	Interval timer.
CLOCK2_LEVEL	Interval timer.
IPI_LEVEL	CPU signalling.
POWER_LEVEL	Power failure.
HIGHEST_LEVEL	Machine-check and bus errors.

The lowest three, PASSIVE_LEVEL to DISPATCH_LEVEL, are utilized for software events; the rest are related in one way or another to hardware events. PASSIVE_LEVEL through DIRQL *n* are part of the execution environment typical of device drivers; the other IRQLs are seldom if ever seen outside core NT code.

Threads

Threads in NT are units of work, under which most routines run. They have priority relative to one another, own resources like timers and are an error boundary (an error under one thread does not necessarily affect another thread). Since threads own resources, thread termination is also the occasion for resource cleanup, such as the closing of files. Threads may be doing work, or they may be waiting for work. Consequently threads can wait on all manner of events (I/O completion, timer objects, objects to be signalled by other threads, for example). OS/390 tasks (TCBs) are analogs of NT threads.

There are, however, a couple of notable differences from OS/390. There are no lightweight threads like SRBs. If a program requires the context (addressability) of a particular thread, the program must create a new thread in the process (address space), or it must queue up an APC to run under the thread. Since the APC suspends a program running under the thread,[23] an APC acts like an IRB routine in OS/390.[24] Also as with IRBs in OS/390, there are situations where an APC will not run under the intended thread.

---

[22] Seen as a scheduling mechanism, the DISPATCH_LEVEL IRQL has an OS/390 analog in the global SRB, which, like a DPC, enjoys higher priority than any task in any address space. But a global SRB runs in a designated address space.

[23] *Windows NT File System Internals*, p. 107.

[24] Note, however, that there is no documented interface for queuing an APC to a thread.

A second difference from OS/390 is that in NT, only threads have priorities; processes do not (but a process' priority class limits how much the priority of threads in the process can change). In OS/390 address spaces have priorities, and within an address space each task or SRB[25] has a priority relative to the other TCBs and SRBs in the address space.

Serialization and coordination

As noted above, IRQL does not ensure serialization on a multi-CPU machine. For kernel-mode routines, spin locks are available. These function rather as do spin locks on OS/390, although NT spin locks do not disable I/O and timer interrupts in themselves but rather do so through their associated IRQL. Spin locks are of course intended to protect relatively short code paths such as updating a queue.

For longer-running code paths under an appropriate (generally this means low) IRQL, a resource object may be acquired with shared or exclusive access, and if it is not immediately available, the invoker may indicate that it will wait until the resource is free. This is rather like ENQ, albeit NT mechanisms do not make possible serialization between machines (as in, say, a GRS ring or a sysplex in OS/390).

For coordination between two programs in NT, synchronization or notification objects may be created, waited on and signalled. If a program needs to wait for a certain time or ensure it is active after a certain time, timer objects may be employed. OS/390's equivalents are the ECB, the associated WAIT and POST services, and timer requests.

Error handling

Every system of any complexity will encounter errors, and a system worth its salt makes error handling a determinate process so that an application—not to speak of the system itself—can intercept errors, retry from them where possible and supply debugging information when not. In both NT and OS/390 error handling is an integral feature.

Corresponding to OS/390's ESTAEX and SETFRR are language-specific mechanisms like _try/_except (C),[26] try/catch (C++) and direct stack-frame manipulation (Assembler). For example,

```
_try
  {
   // begin code coverage

   ...

   // end code coverage
  }
  _except(// filter routine ("recovery" exit)

          ...

        )
   {
    // begin exception handler ("retry")

        ...

    // end exception handler
   }
```

Strictly speaking, the filter routine above is an exit or callback from the C runtime library's error handler ("recovery," in OS/390 terms). The rules the runtime library enforces for a callback are somewhat differ-

[25] This applies to local SRBs. Global SRBs have a priority higher than any TCB for purposes of initial dispatch.
[26] _try, _except and _finally are Microsoft extensions to the C language.

ent from those NT applies to the runtime library's error handler, which NT regards as the "real recovery."[27] In the present case the error-handling rules will be those of the C runtime library except where noted.

The error-handling mechanism illustrated above makes several things possible:

- A range of code (the "try" group) can be covered by specific error-handling code, comprising a filter routine and an exception handler. The filter is like a recovery routine in OS/390.[28]
- Error-handling coverage can be nested, with one or more outer layers of filters, each with a scope designated by a try or equivalent statement.
- Immediately upon occurrence of an error, a kernel-mode exception handler gets control. This handler, acting rather as does RTM in OS/390, builds a description of the error and its context (e.g., the process, the thread, the error-time registers) and, using the description as a parameter, passes control to the runtime library error handler. The runtime handler invokes the lowest-level filter routine as a callback.
- A filter routine can make one of three choices:
    1. It will handle the error. The runtime error handler will give control to the associated exception handler (delimited by the curly braces of the _except statement); the latter, like an OS/390 retry routine, can do fixup and can let control flow down to the mainline.[29]
    2. The error is to be ignored. Then the runtime handler will retry at the point of failure. In such a case, the filter must have ensured, perhaps by altering global or stack variables, that the error will not happen again, lest a recovery/retry loop ensue.
    3. The error is to continue ("percolate") and thus is to be seen and perhaps handled by higher-level filters that the runtime handler is to call.

In OS/390, similar mechanisms of specific coverage, specific handling, nesting and choice of retry or percolation are integral to predictable program behavior.

There are some distinctive features of error handling in NT:

- Error coverage has thread granularity. There is no notion of intra-thread error coverage, such as that for RBs in OS/390. Kernel-mode code, however, has a different error-coverage stack from that for user-mode code.
- For each thread there is a default error-coverage layer, set up when the thread was created. In OS/390 there is of course an analogous default of RTM itself for each RB (and for each SRB); but uniquely in NT, the outermost, default layer can be made to call an exit designated by the application program.
- If an error reaches the default filter, that is, if it is an unhandled exception, the standard action of NT is to terminate the process (not just the thread) where the error occurred. Before termination, however, if an interactive debugger is available, a pop-up offers a human operator the opportunity for debugging. The major exception to debugging followed by process termination is for an error occurring in a service (see below): Here the standard action is not to give a debugger con-

[27] For NT, a real recovery routine is one whose entry point is in a linked chain of error-environment blocks.

[28] The notion of a single real recovery routine and stacked exits from that routine is strikingly similar to the mechanism developed by Peter Relson of IBM OS/390 Development and employed in XCF, ARM, RRS and other places.

[29] An error can be non-continuable. An attempt to handle such an error (non-continuability is a parameter available to the RtlRaiseException and similar APIs) results in an immediate second error, the EXCEPTION_ NONCONTINUABLE_EXCEPTION). OS/390 has a similar attribute for errors, e.g., CALLRTM RETRY=NO.

trol (there may be no console, no human being standing by) but to terminate only the failing thread.

- In kernel-mode execution at DISPATCH_LEVEL IRQL or higher, certain errors (e.g., page faulting, invalid instruction) are not passed to the NT exception handler and thence to the routine's filter routine. Rather, a fatal error ("Blue Screen of Death") occurs. Thus, the ability to handle errors is more limited in some kernel-mode states than it is in similar states in OS/390.
- If a "real-recovery" routine elects not to handle an error and if there is a still older real-recovery routine that does handle the error, the non-handling routine is called by NT a second time, after the handling routine has run but before control flows to whatever retry point the latter selected. This second call is an opportunity for a non-handling routine, especially if it is a runtime library, to do clean-up. This is especially important as the level of error coverage provided by the non-handling routine is about to be removed by NT (it is being "unwound").

Job

In Windows 2000, there is introduced the *job*, which is an object that comprises one or more processes for purposes of management (e.g., priority, user-mode CPU limit and working set minimum and maximum) and security. This has been likened to the Unix process tree.[30] The OS/390 user will be reminded of nothing so much as an WLM enclave, which is a transaction that spans dispatchable units and that is managed as a whole. Perhaps because the concept has been in OS/390 longer than in NT, enclaves have granularity at the level of dispatchable units and not merely of address spaces. It is noteworthy that in both NT and OS/390, it is only relatively recently that an entity spanning several units has been introduced.

Environment subsystems

Windows NT offers several "personalities" via environment subsystems like Win32, POSIX and OS/2. Each is a set of APIs layered on native functions in the Executive; each set of APIs acts to present functions and a run-time environment with a familiar, standard face to developers and users.[31] One finds a similar approach in OS/390, where development and execution environments like CICS and Unix System Services encourage distinctive applications.

A strength of NT's subsystems is the GUI. This presents an immensely more attractive face than does OS/390's 3270 screen. OS/390 can compensate by supporting client applications with a GUI (Web clients supported by servers, for example, or ISPF tricked out with a client-side interface); but client-server programming is inherently more difficult than non-mediated programming.

Both NT and OS/390 are broadening themselves to draw in more ISVs and their applications. NT has seen much development in areas like ActiveX and OLE, whilst the POSIX subsystem has been relatively quiet. OS/390 has experienced an efflorescence of APIs in its Unix System Services; these APIs, along with tools brought to OS/390 from Unix on other platforms, encourage the Unix developer community to write servers or clients for OS/390 or to port to it.

Memory mapping

In Windows NT, two or more processes (address spaces) can share memory by mapping the same storage frames to virtual addresses in each process. If it is a file that the concerned memory range contains, the file is employed for paging rather than the usual paging space. Another feature of mapped memory is that it is subject to the usual Access Control List as other objects. And since the underlying frames and file storage are shared, the processes enjoy a consistent view of the memory's contents.

---

[30] *Inside Windows NT*, pp. 462-464.
[31] *Windows NT File System Internals*, pp. 6-7.

OS/390, for its part, supports memory mapping in general (the IARVSERV service) and for files specifically (the DIV service). IARVSERV is rather versatile, making possible, for example, copy-on-write. The file-mapping capability of DIV, however, is not quite as expansive as file mapping in NT, since only linear VSAM files are supported. And, one notes, IARVSERV was introduced only in recent years.

System activity in the background

Both systems maintain threads or tasks that work in the background. In NT, there are system threads that are created and always run in kernel mode; these are responsible for things like swapping processes out and in, writing changed pages to paging files, workload balancing, I/O caching via read ahead and write behind and handling logon and security requests. Some of these run in the System process, distinguished because it always has process ID 2.[32] OS/390 for its part has special-purpose tasks doing similar duties, executing in authorized state and running in system address spaces, one of which is *MASTER* (always address space ID 1).

Services

These are programs with many of the attributes of OS/390 subsystems and STCs. They are expected to provide functions important or even integral to the proper running of the system. Services typically run with special authority (under the "system account," allowing access to most NT functions and objects), although any valid id/password may be designated as the source of authority. Examples are anti-virus shields, TCP/IP services, the spooler and the logon service. They are on the order of JESx, XCF, CICS, TSO and Unix System Services daemons in OS/390.

Services are usually started automatically, fairly late in the boot process but before any human operator has a chance to log on. A service's start-up can be made contingent on the start-up of other services. In fact, successful completion of NT's own start-up can be made dependent on successful start-up of the service. Alternatively, starting a service can be manual (by a person in a command window or via a GUI, or by another program employing an API).

A service runs in a process, which it may share with other services if all are packaged in a single executable as separate functions. Each service has its own thread. Usually a service has no GUI or other user interface. The service is rather intended to accept requests via a programming interface such as a named pipe.

A single NT component, the Service Control Manager (SCM), is the conduit for requests to start a service. SCM is further responsible for passing along control directives such as pause (suspend function), resume and stop. SCM is "network-aware," allowing control of machines other than the local one. Thus, SCM discharges responsibilities that are performed in OS/390 variously by the Master subsystem, JESx, CONSOLE and XCF.

One other major function of a service is to establish a device driver dynamically (after boot time). This is done by a program (with sufficient access rights) opening a session with SCM, defining a service whose executable is the driver, and starting the service. (Note, by the way, that the driver persists even if the opening program closes its SCM session and terminates, and that no process or thread is created for the driver.)

Object versus control block

In NT, an object is some resource that can be manipulated by associated routines (perhaps in a runtime library) but which is not otherwise externalized. In OS/390, such an entity is a control block available

[32] *Inside Windows NT*, p. 71.

through associated services. Timers—timer objects in NT along with routines like KeSetTimer, TQEs (timer queue elements) in conjunction with STIMERM in OS/390—are cases in point. NT is unquestionably richer in objects, but in OS/390 one sees ever more of them, for example, name/token services, XCF services, sysplex (Coupling Facility) services and RRS services in successive SCP releases.

The registry

The Windows NT registry has a role unlike that of any single mechanism in OS/390. It keeps configuration information and so is used at boot time, rather like IODF datasets and SYS1.PARMLIB by OS/390. The registry holds information about installed software, including any state information the software may wish to save. The registry contains user information and access information, like RACF, Top Secret or other security products (security functions in NT can be supplemented by authentication packages). Hence the registry is something of a general-purpose repository, capable of storing whatever information can be made to fit into its directory-like structure.

The registry has some interesting features. For example, a piece of the registry can be protected from access to programs (or users) who lack sufficient privilege. By default some pieces are so protected, and other pieces are open to all. Another characteristic of the registry is that a program can be apprised of a change to a piece of the registry by opening a handle to the piece ("key") of interest, associating a notification event for the key and waiting on the event. A third feature is that the registry can be backed up, in whole or in part (by its constituent "hives," which are the files that are the registry's backing, or even by its keys, which are arranged in a tree-like, directory structure). Given the popularity of registry hacks, the flexible capability of registry backup is no small virtue.

Development and Usability Issues

From the perspective of a programmer, the two systems feel different. Each system possesses characteristics not easily related to underlying structure but important in their own right, for they affect how the designer lays out his product and how the developer makes use of, or struggles with, the system in creating the product.

Reliability

Windows NT's capability for handling, isolating and recovering from errors differs from OS/390's. It is the author's experience in using NT as a desktop system and as a development and test platform that NT is more liable to failure as a result of errors. OS/390 seems more resilient, more able to recover or to degrade in function in a controlled manner. To the extent that Windows NT and OS/390 do exhibit different characteristics in handling errors, that may be not because one or the other is less well designed and implemented, less mature or "less good." Rather, each OS has goals peculiar to it and distinctive approaches to those goals.

Since day 1 (the early 1970s), OS/390 and its predecessor versions of MVS have had the goal of confining the effects of errors to the routines experiencing (if not introducing) them. VM, an IBM operating system with a history at least as long as that of MVS, has always taken a different path, which often means preserving system and data integrity so much as possible, going belly-up and being re-IPLed quickly. In respect to reliability, Windows NT seems more like VM than like OS/390.

And, lest we forget, it took MVS more than a few years before it achieved the robustness that we expect of it today.

Systems-programming language

In comparison with things like I/O handling and scheduling, programming languages are a peripheral feature of the two operating systems. Yet if peripheral, languages are critical to creating "system-type" products. C is the language in which NT itself is written (for the most part). PL/X has that role in OS/390 (again, mostly). PL/X, albeit not available to all and sundry, has been provided to some ISVs for development; others, and the many installations that have narrow-scope code like exits, make do with Assembler. Although rating languages is like comparing tastes in music, the author (an apprentice C programmer) finds PL/X more supple. On the other hand, C's try-except-finally implementation of structured exception handling gives the language an edge in error handling when compared to the footprinting and recovery/retry techniques one employs in PL/X or, for that matter, in Assembler.

Development platform

Perhaps because the GUI is integral to NT, that system is fairly easy to use as a development platform. Although NT is a complex system, its interactive nature, when coupled with a good source-code debugger (the author has found NuMega's SoftICE quite powerful), makes it possible to overcome the complexity quickly. Microsoft has been criticized for fostering closed programming environments, yet in NT there is a wealth of information and material (e.g., numerous code samples), especially in the Microsoft Developer Network's Device Driver Kit and in the Microsoft Knowledge Base. There is further a flourishing developer community of ISVs, who exchange advice, insight and tips.

OS/390 is a continent, and knowledge and tools have historically been concentrated in IBM. Happily the company is more open than once it was, its developers are more accessible, and ISVs are courted.[33] But if

[33] See, for instance, the set of Component Broker pages at http://www.software.ibm.com/ad/cb.

probably not a great deal more complex than what NT has come to be, OS/390 is at base less forgiving in development and testing. Only if an OS/390 developer has ready access to developers within IBM and can manage interactive testing through VM (to run an OS/390 image as guest) can he or she feel on equal footing with the NT programmer who can draw on the DDK, ISV newsgroups and SoftICE.

Conclusions

The list of similarities above could have been longer. A few more differences might have been adduced. Coming from different backgrounds and with aims peculiar to each, OS/390 and Windows NT are involved, highly evolved systems with quite a few mechanisms that look alike. I/O handling is the closest parallel. Virtual addressability turns out to be very close, as does the priority-oriented, interrupt-driven scheduler or dispatcher (to use NT's and OS/390's respective terms). Yet scheduling is also the big difference: IRQL in NT is not like anything in OS/390, and OS/390 has more execution states for its dispatcher to handle, what with SRBs, cross- and multiple-memory addressability, recovery environment and so forth. That said, both systems are at base concerned with expediting units of work and handling interruptions generated directly or indirectly by that work, and things like IRQL, number of states and GUI can be seen as comparative details.

Regarded as multiuser and server environments, the two systems have come to be much like each other. Are they brothers? No. But probably cousins, one or two degrees removed.

References

Steve Ballmer, http://www.microsoft.com/presspass/exec/steve/9-28bizapps.htm. Size of NT (whether NT 4 or Windows 2000 wasn't clear in the context).

"A Crash Course on the Depths of Win32 Structured Exception Handling," Matt Pietrek, *Microsoft Systems Journal*, January 1997. (Almost) all you'd ever want to know about error handling in WinNT.

Computerworld Online News, 06/09/98 05:34 PM, http://www.computerworld.com/home/news.nsf/all/9806092nt. Report of the Giga Information Group estimate of NT 5.0's size.

Ctrl2cap is a sample device driver available at http://www.sysinternals.com. RegMon at the same site includes Instdrv, a sample routine for installing a device driver.

Inside Windows NT (second edition), David A. Solomon. Microsoft Press, 1998. A wide-ranging survey of NT, including many internals.

Intel Architecture Software Developer's Manual, volume 3: System Programming Guide, order number 243192 (this series of manuals pertains to Pentium II). Intel Corporation, 1997. The architecture and the instructions.

Microsoft Windows NT Device Driver Kit (DDK), version 4.0; this is part of the Microsoft Developer Network subscription. Specification of kernel functions; numerous examples.

Showstopper!, G. Pascal Zachary. Free Press, 1994. Blow-by-blow account of the development of Windows NT 3.0.

SoftICE for Windows NT, v3.24, a product of NuMega Technologies, Inc. See http://www.numega.com.

"Windows NT and VMS: The Rest of the Story" and "NT vs. UNIX: Is One Substantially Better?," Mark Russinovich, *Windows NT Magazine*, December 1998.

Windows NT Device Driver Development, Peter G. Viscarola and W. Anthony Mason. Macmillan, 1998. Exceptionally thorough introduction to Windows NT and to writing device drivers.

Windows NT File System Internals, Rajeev Nagar. O'Reilly, 1997. Very good introduction to Windows NT and to writing device drivers.

ZDNet, http://www.zdnet.com/pcmag/insites/ dvorak/jd981116.htm. Report of Gartner Group estimate of NT 5.0's size.