Portable Systems Group

NT OS/2 Subsystem Design Rationale

Author: *Mark H. Lucovsky*

Revision 1.3, June 1, 1989
Original Draft, May 26, 1989

1. The NT OS/2 Mission

The **NT OS/2** group was formed with a clear mission:

- To design and implement an OS/2-compatible operating system for non-x86 hardware platforms
- To support the APIs required by POSIX (IEEE Std 1003.1-1988) at a level required to pass government validation
- To support symmetric multiprocessing
- To provide C2 security features with a path to B1 and beyond
- To provide easy portability to other 32-bit architectures
- To design and implement the first functional system by the 3rd quarter of 1990
- To target the system for a **Microsoft**-designed i860 PC hardware platform, followed shortly thereafter by an i860mp or N11 multi-processor server system

Conclusions from the January 1989 System Retreat indicated that **NT OS/2** is critical to the long-term growth of **Microsoft**. The design of the system must accommodate current and future needs of **Microsoft**. The design must be maintainable, and easily extensible.

2. Design Goals

In order to achieve our mission, the following set of prioritized goals was established:

1. Robustness. The highest priority for **NT OS/2** is robustness. The inner workings of the system should be straightforward and well defined. A complete and formal design on all components of the system must be produced and interfaces and behavior must be well specified. The system must be designed without "magic".

2. Extensibility and maintainability. **NT OS/2** must be designed with the future in mind. It should be easily extensible to meet the needs of our OEM customers and our own needs over time. The system should also be designed for maintainability.

   Given the state of the API sets that **NT OS/2** must support, its design must accommodate changes and future additions to those sets.

3. Portability. **NT OS/2** must be designed for portability. The system architecture must be portable across a number of platforms. There are portions of the actual implementation that will require a port when moving from platform to platform. The effort required to port **NT OS/2** from one platform to another must be less than, or equal to, an equivalent port of a UNIX or Mach system.

4. Performance. Superior performance in **NT OS/2** is important. Algorithms and data structures that will lead to a high level of performance and that will provide us with the flexibility needed to achieve our other goals must be incorporated into the design. The granularity of locking, the various types of locks used in the system, the amount of time spent at an elevated interrupt level or with interrupts completely disabled must be carefully designed so that **NT OS/2** is a responsive system which can compete in a number of markets.

In addition to these goals, compatibility with OS/2 APIs and POSIX compliance are system constraints in **NT OS/2**.

3. Design Alternatives Investigated

Several design alternatives for **NT OS/2** were considered during the design phase.

The first design layered the POSIX API set on top of a slightly extended OS/2 API set. As the design progressed, it became apparent that this design would lead to a system that could not achieve the goals of robustness, maintainability, or extensibility. Problems encountered with a similar attempt in OS/2 led to considerable change in the base system capabilities, which further strengthened the belief that this was a poor alternative.

The next design implemented both OS/2 and POSIX API sets directly in the **NT OS/2** executive. This was an improvement on the previous design, but the large number of "chicken wire" and "voodoo" interfaces required by this design threatened the goals of extensibility and maintainability.

The third design implemented OS/2 and POSIX as protected subsystems outside the **NT OS/2** executive. Success with this type of client/server architecture in the academic community and at other research sites provides strong evidence that this design will allow **NT OS/2** to meet its goals of robustness, extensibility, maintainability, portability, and performance, and thus, achieve its mission. Therefore, this design was chosen for **NT OS/2**.

(The final section of this document examines the three **NT OS/2** design alternatives in greater detail.)

4. The NT OS/2 Design

The **NT OS/2** system design consists of a highly functional executive, which executes in kernel mode, and exports a native API (a set of system services). Operating system environments such as OS/2 and POSIX are implemented as protected subsystems outside the executive.

A protected subsystem executes in user mode as a regular (native) process. The subsystem may have amplified privileges, but it is not considered a part of the executive and, therefore, cannot bypass the system security architecture, or in any other way corrupt the system. Subsystems communicate with their clients and each other using a high-performance local (cross-process) procedure call, or LPC, mechanism. (A round-trip LPC completes in approximately 100usec on the i860.)

This **NT OS/2** design satisfies each of the goals for the system. The following attributes of the design ensure the primary goal of robustness:

- The kernel mode portion of the system exports well-defined APIs that, in general, do not have mode parameters or other "magical flags". Therefore, the APIs are simple to implement, easy to test, and easy to document.

- A formal design is being produced for all portions of the **NT OS/2** system prior to coding. This effort has led to well-documented interfaces for native services and internal functions.

- The partitioning of major components, such as PM, OS/2, and POSIX, into separate subsystems is resulting in simple, elegant designs in the subsystems. Each subsystem is optimized to implement only those features needed to provide its API set.

- With the prevalent use of frame-based exception handlers, **NT OS/2** and its subsystems are able to catch programming errors and filter bad or inaccessible parameters in an efficient and reliable manner.

The **NT OS/2** design also meets its goals of maintainability and extensibility through the following features:

- The **NT OS/2** design is simple and well documented. This, coupled with a common coding standard used throughout the system, should enable a programmer to work on any piece of the system without having to consult the "gurus" to learn about hidden rules, side effects, or "magical" programming tricks.

- By using subsystems to implement major portions of the system, **NT OS/2** isolates and controls dependencies. For example, the only piece of the **NT OS/2** system affected by the changing Cruiser design is the OS/2 subsystem. The design of the process structure, memory management, synchronization primitives, and so on, does not have to be put on hold. The same holds true for the evolving POSIX standards.

- As the needs of **Microsoft** grow, the **NT OS/2** system is prepared to accommodate those needs. Subsystems that provide additional functionality can be added to the system without impacting the base system. New subsystems can be added without having to modify the **NT OS/2** executive or release a new version of the system.

  Subsystems such as DOS, Windows, or Xenix can be added to the system if necessary. OEMs could continue to provide limited support for operating system environments other than the **Microsoft**-provided OS/2 and POSIX environments.

- Using the subsystem or "building block" approach, it is possible to envision a configuration that includes only the OS/2 subsystem. POSIX could be a revenue-producing, licensable option. If the option were not used, no system resources would be sacrificed.

- Subsystems need not bypass the security features present in **NT OS/2**. Rather, they can use the security features to their fullest extent.

NT OS/2 portability is ensured by the following:

- Except for small, well-isolated sections of code, **NT OS/2** is written in C. The system is being developed on prototype compilers with limited functionality, and still, the design has yielded portable code.

- Using the UNIX and Mach porting experience of engineers on the project, the group has established that the **NT OS/2** will port to other platforms at least as easily as the UNIX or Mach operating systems. The effort involved in porting **NT OS/2** to another 32-bit, paged architecture, using readily available compilers, is small.

NT OS/2 is a high-performance system designed to run on high-performance hardware. We believe that the system will perform better than any system providing equivalent functionality on equivalent hardware. The following attributes of the system promote high performance:

- Algorithms and execution paths through the system have been carefully optimized to increase performance. Also, the modular nature of the system allows performance optimization by replacing entire components.

- System calls, exceptions (page faults), LPC, thread creation, and I/O have undergone scrutiny to ensure their speed. The round-trip time for a null system call is currently on the order of 3usec (on a 40Mhz i860). Given this number, **NT OS/2** performs better than most systems even after equalizing processor speeds.

- Ensuring high performance is an ongoing activity in the implementation of **NT OS/2**.

5. Performance in the Subsystem Model

Before committing the **NT OS/2** design to a subsystem, or client/server model, time was spent analyzing the Presentation Manager. One of the deficiencies in the current implementation of PM is that it must manage global state without having any way to protect the state. We worked with one of the designers and implementors of PM to develop a solution to this problem by making PM a protected subsystem (which executes in its own process context rather than in the context of the thread that called a PM entry point).

Before proceeding with the PM design, the **NT OS/2** LPC mechanism was designed. We felt that if the LPC design were solid, it could be modeled, and we could determine whether or not PM performance would be acceptable using a subsystem design model.

Ideas present in several high-performance LPC mechanisms were incorporated into the **NT OS/2** design:

- The ability to efficiently pass small amounts of data, as was done in Stanford's V system, is included.

- The idea of mapping large messages or passing large parameters "out-of-band" is similar to the mechanism used in Carnegie Mellon's Mach system.

- The ability to pass message data through memory shared between the client application and the subsystem is similar to the technique used in an experimental system under development at the University of Washington, and which also appears in DEC's Topaz system.

With the design of the **NT OS/2** LPC mechanism complete, a model was created to measure the performance impact of running PM as a protected subsystem.

The model consisted of the following pieces of modified system software:

- OS/2 Kernel Modifications. A special version of OS/2 1.1 was built. This version of the system had an additional system service that simulated a context switch from the calling thread back to the calling thread.* All of the work involved in switching address spaces was simulated as well.

- *pmwin.dll* and *pmgpi.dll*. A new version of each of these libraries was created. For each entry point, the cost of marshalling its parameters into and out of a message buffer was simulated; two calls to the new context switch routine were done; and finally, a call was made to the original version of the entry point.

By running PM applications using the modified system software, we were able to determine exactly how much overhead PM would incur when run as a subsystem.

Several test cases were run on the model. These included running the PMBENCH benchmark suite, running PMDRAW and drawing complicated pictures, running various configurations of PM Excel and scrolling, drawing charts, and performing other screen manipulations, and finally, running a journaled interactive session with multiple PM applications doing different tasks, including menu and dialog box operations.

Before running our tests, we did not know what to expect. We felt that if the system did not feel sluggish, then the subsystem approach might be acceptable. After running all of our tests, we were surprised. The system performed so well that we could not tell the difference between the subsystem version of PM and the normal version of PM.

The following table shows a condensed listing of our benchmark results:

	LPC PM	Standard	Subsystem	
	Overhead	PM Time	PM Time	Difference

* This simulation involved invalidating mapping information, saving and restoring registers, and saving and restoring the mapping information.

PMBENCH Test Suite	5.14%	***	***	***
PMDRAW monticello	16.88%	12.403s	14.497s	2.094s
PMDRAW fish	8.80%	11.887s	12.940s	1.053s
Excel Scroll 1's	3.25%	30.880s	31.885s	1.005s
Excel Scroll Big	0.84%	63.060s	63.590s	0.530s
Excel Chart	9.65%	12.900s	14.145s	1.245s
Interactive	1.20%	335.510s	339.670s	4.160s
				=======
Average Overhead	2.16%	466.640s	476.727s	10.087s

From the results of our study, we felt that the additional overhead imposed by running PM as a protected subsystem was acceptable given the benefits of such a design. While there is measurable overhead, it is not detectable when sitting in front of a machine running interactive or graphics-intensive applications.

After determining that PM could be run as a protected subsystem without incurring unacceptable performance degradation, we looked at other areas of the system that would be cleaner to implement as a separate subsystem but would not impact overall system performance.

Given that OS/2 and POSIX had to be treated as partitioned code within the executive, they were natural candidates for implementation as protected subsystems. We believe that real OS/2 (and POSIX) applications will be more dependent on the performance of PM than any other portion of the system. The ratio of PM to operating system service calls is likely to range from 10:1 to 100:1. If PM is a good candidate for implementation as a protected subsystem, then operating system environments such as OS/2 or POSIX are also good (if not better) candidates.

6. Standards

During the initial design phase of **NT OS/2**, a great deal of time was spent examining ways to design a system that could support both the OS/2 and POSIX API sets. This job was complicated by the fact that both of the API sets we planned to support were moving targets. In fact, the Cruiser specification was not yet available; it is still evolving.

6.1 OS/2 Standards

Our initial OS/2 API set centers around the evolving 32-bit Cruiser, or OS/2 2.0 API set. (The design of Cruiser APIs is being done in parallel with the **NT OS/2** design.) In some respects, this standard is harder to deal with than the POSIX standards. OS/2 is tied to the Intel x86 architecture and these dependencies show up in a number of APIs. Given the nature of OS/2 design (the joint development agreement), we have had little success in influencing the design of the 2.0 APIs so that they are portable and reasonable to implement on non-x86 systems. In addition, the issue of binary compatibility with OS/2 arises when the system is back-ported to an 80386 platform. This may involve 16-bit as well as 32-bit binary compatibility.

6.2 POSIX Standards

Our initial POSIX efforts center around the IEEE Std 1003.1-1988 (or Draft 13). The spec is vague in several areas and contains several optional features.

In order to sell in certain federal government markets, a POSIX implementation must be compliant with FIPS 151. This FIPS requires that certain optional features of POSIX be implemented, and also requires portions of other POSIX standards (1003.2, "Applications and Utilities"). In addition, the FIPS requires a certification of conformance. This certificate can be obtained by passing a certified POSIX test suite. The current set of test suites are developed by third parties, and do test for compliance with the POSIX spec. Unfortunately for us, the test suites were developed on UNIX systems that claim POSIX compliance. The test suites end up testing a lot of UNIX folklore that happens to be permissible under an interpretation of the POSIX spec.

To further complicate POSIX compliance, additional drafts of 1003.1, which are close to approval, have been proposed. The effects of approval are unknown. It is not clear if future additions to POSIX will be required under future FIPS, or if additions will be made optional. The government standards body that is issuing the FIPS is apparently ready to add any approved POSIX drafts to its FIPS. The latest draft under consideration (1003.1a), would add a number of features from Berkeley UNIX 4.3 to POSIX. It is anticipated that a new FIPS will be issued which requires these features in order to participate in certain government markets.

7. An Analysis of the Design Alternatives

Once the mission and goals of **NT OS/2** were clear, the design work was started. The most difficult portion of the design centered around the issue of how to provide OS/2 and POSIX compliance on the same system without failing to achieve our mission or compromising our goals.

Combining the APIs of multiple operating systems in a single system is always a difficult task. It does not matter whether the APIs are similar or different. The most striking example of this problem is the poor integration of UNIX variants found in the current UNIX market.

In the beginning (1982-1984), there were basically two branches in the UNIX tree. The BSD branch with Berkeley UNIX 4.2 and 4.3, and the AT&T System V branch with System V.2 and V.3.

Companies that offered pure systems in either camp were the norm. Companies in the scientific and engineering markets supported BSD while business-oriented companies supported System V:

- Sun 1.0-2.x was pure BSD
- DEC's ULTRIX was pure BSD
- Sequent was pure System V
- Altos was pure System V

After some time, companies began to offer systems with mixed features. This began with systems advertising "System V with BSD networking." Soon, nearly all companies offered systems with some features from both environments. Applications could call APIs from either set. If the API specified different behavior for a System V or a BSD implementation, it was usually a tossup as to which semantics were followed.

The current state of System V and BSD integration is the root of nearly all the confusion in the current UNIX marketplace. To port an application that was originally BSD to a system that is "System V with BSD features" requires elaborate configuration files that "pick and choose" the APIs. With each port to a new system, the configuration options and combinations must be expanded to accommodate the new system. The popular UNIX editor, emacs, is a perfect example of this. The emacs editor comes with nearly 50 configuration files. Each file describes a derivative of UNIX that has different features and supports a certain mix of BSD and System V APIs.

A major design issue in **NT OS/2** is to avoid the integration-of-features problem present in the current UNIX marketplace. **Microsoft** cannot afford to present POSIX and OS/2 integration as poorly as most of the UNIX vendors have.

In the selected **NT OS/2** design, an application that uses OS/2 APIs may only use OS/2 APIs. The POSIX API set is not available to the application. The reverse restriction is also true. POSIX applications may not call OS/2 APIs.

7.1 POSIX Layered on OS/2

The first alternative examined the feasibility of layering the POSIX API set as a runtime package on top of a native system service interface based on an OS/2 API set.

Using this approach, the **NT OS/2** executive would export an OS/2 2.0 API set. If there were functions that required extensions in order to make this work, we were prepared to make those extensions. An example of this approach is supporting POSIX *fork()* and *exec()* using OS/2's DosExecPgm().

We proposed adding a flag to DosExecPgm that would take one of the following values:

1. The API should work exactly as the current DosExecPgm function works (that is, a new process is created and its address space is initialized so that it maps the image specified as the program name parameter).

2 The API should create a process and the address space should be an image of the address space of the calling process. Thread 1 should be created in the new process and its initial context should be identical to the context of the calling thread at the time of the call. The only exception is that thread 1 in the new process must return with a different return value than that returned by the calling thread.

3 The API should clean the address space of the process, terminate any threads in the process, create a new address space such that it maps the specified program image file, and create thread 1 so that it begins execution at the entry point specified in the image.

To implement OS/2 DosExecPgm, the API would be called with flag value 1. POSIX's *fork()* and *exec()* would be implemented using flag values 2 and 3.

On the surface, the above technique seems to work, but it is complicated. Complications arise in the following areas:

o File descriptors owned by a process would be dealt with differently in all three variations of DosExecPgm().

o File locks held by the process at the time of the API call would be handled differently for all three cases. In fact, since file locking itself is different, the case is really an 8-way case.

o Outstanding timers or process alarms have at least three different actions.

o Signals pending, or the state of a process's signal or exception handlers, is affected by the various API options.

The list of problems with this API is large, as should be clear from the above list. More important, the problem seems to scale exponentially. Simple operations like opening or creating files, establishing signal or exception handlers, reading from and writing to the terminal, or even manipulating regular files all have problems and virtually all require a mode argument.

One of the other serious problems with this design alternative is that it presents a poor integration of OS/2 and POSIX. It would be difficult to separate OS/2 calls from POSIX calls. Multi-threaded OS/2 applications that, either on purpose or as a result of a programming error, call DosExecPgm specifying a POSIX-oriented option would have disastrous effects. We could always say that this could not happen, but in order to achieve the robustness goals of the **NT OS/2** system, the executive would have to be coded so that it could handle all possible incorrect parameter combinations.

After determining that layering POSIX on top of OS/2 would bury much of POSIX in the executive, and would cause most of the overlapping APIs to require a mode parameter, we looked at ways of implementing the POSIX API set directly inside the **NT OS/2** executive.

7.2 OS/2 and POSIX in the Executive

By implementing both the OS/2 and POSIX API sets directly within the executive, we were able to work on a layered, controllable design. The system would yield two API layers, one layer exporting OS/2 APIs and the other layer exporting POSIX APIs. The API layers would be implemented on top of an executive support layer.

The executive support layer would implement basic executive services such as process and address space management, thread creation/deletion/control, security, an I/O system and a file system. The executive support layer would control, create, and delete all state in the system. The API layers would simply call the executive support layer with appropriate parameters. They would not maintain state.

As we progressed with this design, it became clear that it was nearly identical to our initial design. Our proposals for the design of the process structure were not much different from the extensions that we had planned for DosExecPgm(). The primary difference was that the parameter combinations passed to the executive layer were controllable. Since the parameters came from the system code that implemented the API layers, we were able to make rules and declare that certain parameter combinations could not occur. This made the executive layer somewhat easier to write, but the rules for calling the executive became rather elaborate.

For **NT OS/2** to remain a product that could carry **Microsoft** through the 1990's, maintainability, extensibility, and robustness had to be ensured. It seemed that almost everything became an exception. The well-defined interfaces within the process structure became littered with exceptions and kludges needed to support the demands of POSIX's job control option or OS/2's complex process/command subtree relationships. Simple functions, such as waiting on a child process (common to both OS/2 and POSIX), became difficult to implement because the executive had to manage two slightly different cases.

As each new issue arose, the solution always seemed to have a common theme...

The terminal driver could look to see if the application writing to the terminal was a POSIX application. If so, then if the terminal was not the controlling terminal for the process, but the process was not ignoring SIGSTOP, then the process could be signaled and its parent notified.

or...

When a process terminates, look to see if it was an OS/2 application or if it was a POSIX application. If it was an OS/2 application that was *exec*'d using EXEC_SYNC, then after termination is complete, the process ID is available for re-use. If it was a POSIX application, then if the parent was not PID 1, signal it. If the process was a session group leader, then generate a SIGHUP signal to all members of the session group with the same controlling terminal, and possibly free the controlling terminal.

The more the design progressed, the more the system started to look like a bowl of spaghetti. Problems arose due to subtle differences between OS/2 and POSIX in almost all areas. The following are a few examples of the problems:

- o Process ID (PID). The POSIX job control option (required by FIPS 151) is difficult to implement correctly even on a BSD UNIX system. Process relationships and the lifetime of a PID are complex. A POSIX PID has nothing in common with an OS/2 PID other than sharing the same acronym.

  The standard solution to this sort of problem usually involved a "table off to the side" that could keep track of the differences. We had "tables off to the side" for POSIX and OS/2 process IDs, POSIX sessions, job control sessions, controlling terminal IDs, file and file system serial numbers (device, inode pairs, etc.), and others.

- o Exception handling. POSIX requires an exception handling mechanism based on signals that are similar to signals found in Berkeley UNIX 4.3. This architecture is drastically different from the current 16-bit OS/2 exception architecture and even more different than portions of the proposed OS/2 2.0 exception architecture.

  The exception architectures of both systems involve large portions of the entire system. The keyboard, video, and terminal drivers are involved, as is the process structure, system service dispatcher, trap handler, and so on.

  Trying to tie together these different pieces of the system in a way in which they could all participate in exception handling was seriously compromising the design of the system.

  The solution to this sort of problem usually involved adding fields to the process or thread structures to keep track of this. It became clear that our process and thread structures were going to be large. Much of the overhead was due to link words and pointers to the "tables off to the side," or to fields that were needed only if the process or thread represented a POSIX application (or OS/2 application).

- o Security. POSIX security impacts major pieces of the system. As the design progressed, it became clear that POSIX security was at odds with the Cruiser-like security scheme being designed for **NT OS/2**. Many features of the security scheme would have to be bypassed in order to implement the "hodge podge" of security features/APIs that appear in POSIX.

The list of chicken wire fixes is endless. Nearly all areas of the system are involved, including timers, time-of-day format, file locks, pipes, and many others.

The only advantage that this solution had over the previous one was that the API layer could call the executive support layer with a known set of parameter combinations. The executive support layer did not have to deal with illegal parameter combinations.

NT OS/2 had to explore some new alternatives. What we needed was a mechanism that would allow the OS/2 API layer to manage and control all state for all of the OS/2 applications in the system, and to allow the POSIX API layer to do the same for all of its applications. It was this realization that brought us to the current design strategy for **NT OS/2**.

7.3 POSIX and OS/2 as Subsystems

The system architecture chosen for **NT OS/2** allows it to achieve its goals and, therefore, fulfill its mission. **NT OS/2** is designed with a small, non-preemptible kernel, which executes in kernel mode. A small but highly functional, preemptible, interruptible, and reentrant executive, which also executes in kernel mode and which exports a number of system service APIs, is layered on top of the kernel.

The APIs exported by the executive do not implement either the OS/2 or POSIX API sets. Instead, they export a set of APIs that allow both an OS/2 API set and a POSIX API set to be implemented entirely in user mode as separate processes running as protected subsystems. Using this approach, an OS/2 or POSIX API is emulated using the following sequence:

- An application calls the local stub for an API function.

- The stub packages the arguments into a message and transmits the message to either an OS/2 or POSIX subsystem using the **NT OS/2** local procedure call mechanism.

- The subsystem receives the message, implements the API, and replies to the application using LPC.

- The local stub receives the reply and returns the results to the application.

The APIs exported by the **NT OS/2** executive are powerful, but at the same time, are simple and straightforward. There are no cases in which a single flag parameter changes the entire meaning of an API. This design technique allows **NT OS/2** to achieve its goals of robustness, extensibility, and maintainability.

Implementing OS/2 and POSIX as subsystems allows each subsystem to implement only the set of semantics required by that subsystem. The requirements of the subsystems do not translate into "tables off to the side" or extra fields in data structures managed by the executive. When a subsystem needs to keep track of additional state associated with an object, it does so in its own data structures managed in the address space of the subsystem. This technique leads to more elegant solutions to problems posed by OS/2's process relationships or by POSIX's job control data structures.

Rather than having to bypass most of the security features present in **NT OS/2**, subsystems are able to use the security features to their fullest extent. The security architecture, along with the high performance LPC mechanism and powerful process structure and memory management APIs allow the subsystems to increase the robustness, extensibility and maintainability of the system while at the same time decreasing the demands on system resources.