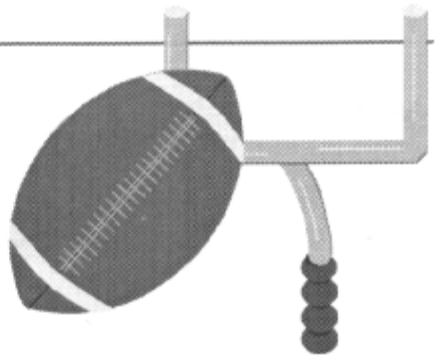
The Goals

- **Extensibility**
- **Portability**
- **Robustness**
- **Compatibility**
- **Multi-processing**
- **Distributed Computing**
- **POSIX compliance**
- **Security**
- **Performance**

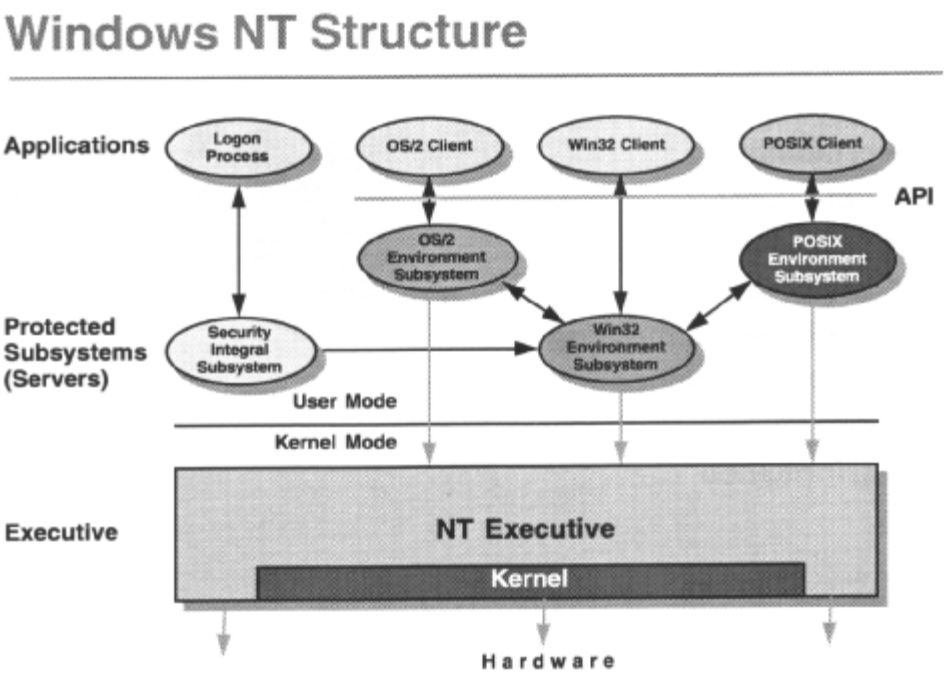


Windows NT is designed to address the changing requirements of the computing world. It is written mainly in C and is crafted in such a way as to make its functionality extensible, and to ease the porting of the code from one hardware platform to another. This enables the ability to take advantage of multiprocessor and RISC computers, and to distribute tasks to other computers on the network, transparently.

Whilst providing applications and users with the ability to use the power of local and remote machines, Windows NT must offer compatibility to applications and users. Users must feel comfortable with the interface, and be able to run existing high-volume applications. Existing applications have to port simply to the new environment to take advantage of its power. So, the user interface is compatible with existing Microsoft systems and existing programming APIs are supported and have been extended.

To be considered a major player in the server arena, Windows NT has to offer reliable, robust support for '*mission critical*' software. This means the system should be fault tolerant, protecting itself from malfunction and from external tampering. It should behave predictably and applications should not be able to adversely affect the system or each other. It should also have a security policy to protect the use of system resources, and implement resource quotas and auditing. Networking is built in, with high level programming and user interfaces available. Remote access to other machines on various networks is almost transparent.

Because applications have to perform to an expected level, the system should be fast and responsive on each hardware platform.

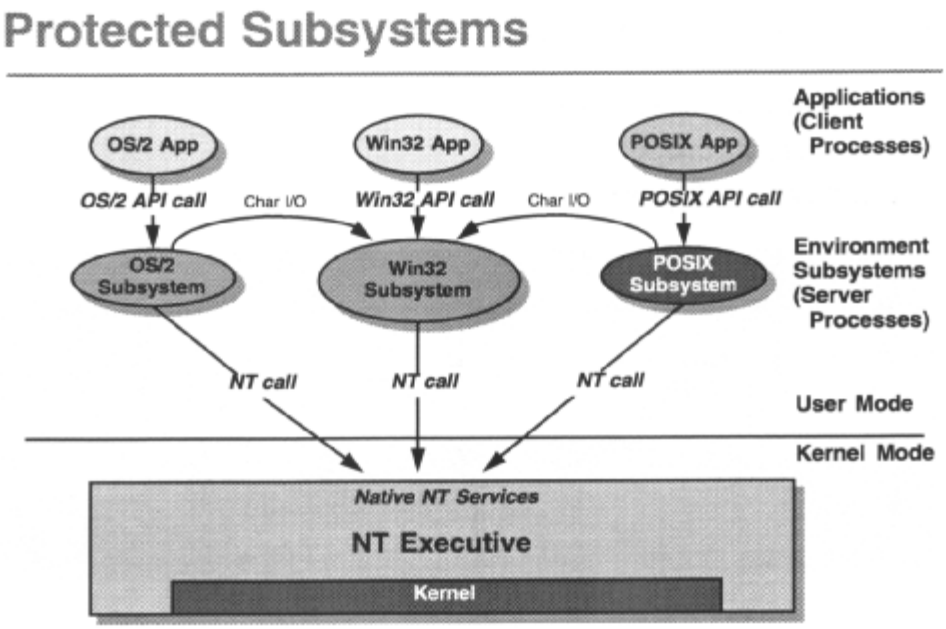


The structure of Windows NT can be divided into two parts: *user-mode* and *kernel-mode* (sometimes called supervisor-mode on other operating systems). The user-mode part contains a number of protected *subsystems*, so-called because each one is a separate process with its own protected virtual address space. The NT *Executive* is the kernel-mode part of Windows NT, which exports generic services that protected subsystems can call to obtain basic operating system services. Apart from the lack of a user interface, the Executive is a complete self-contained operating system in its own right.

There are two types of protected subsystem: *environment subsystems* and *integral subsystems*. An environment subsystem is server process that provides a particular view of Windows NT to a client application, by emulating a particular operating system environment and providing a specific Application Programming Interface (API). Current environment subsystems supported are Win32, OS/2 and POSIX. Integral subsystems are servers that perform other important operating system functions. For example, the Security Subsystem is implemented as a user-mode integral subsystem.

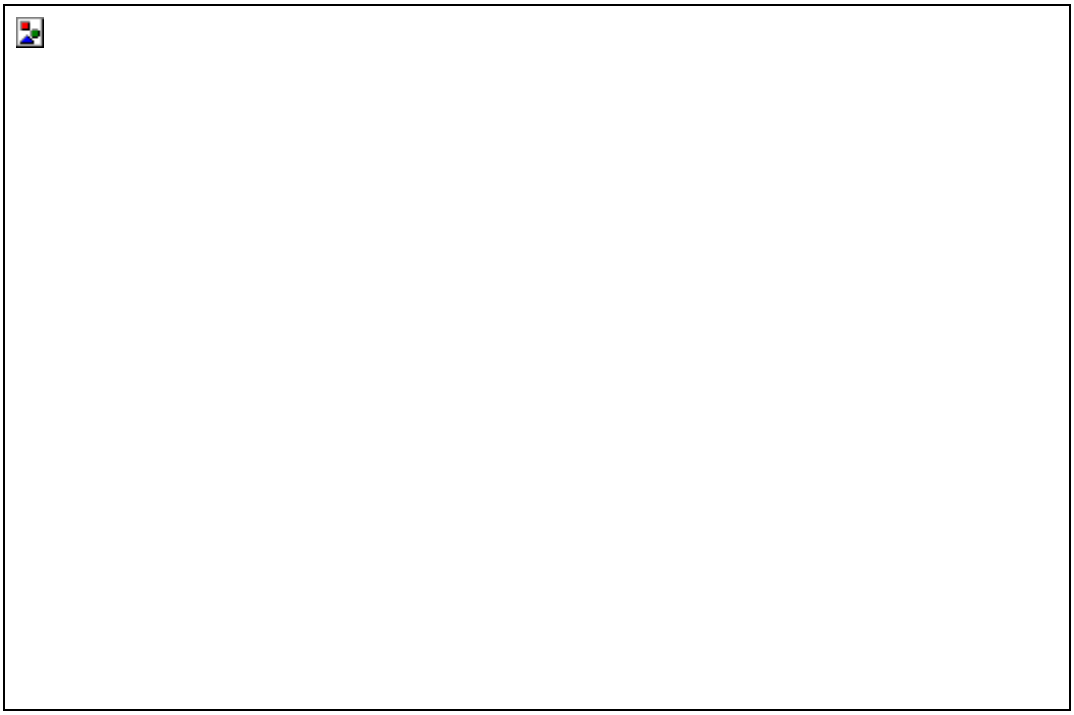
The most important environment subsystem is the Win32 subsystem that supplies the Win32 API to 32-bit Windows applications. It also provides mouse and keyboard input and both graphical and character-mode screen output for all environment subsystems. In other words, all interaction with the user is controlled by the Win32 subsystem, thus allowing many types of applications to run seamlessly on the same desktop. The OS/2 environment only supports 16-bit character-mode applications (i.e. not PM applications) and is only supported on Intel x86-based computers. However, real mode OS/2 applications can run on non-Intel x86-based computers in the MS-DOS environment. The POSIX subsystem meets the requirements of the IEEE 10003.1 standard - a C programming interface to UNIX, which is of limited use but required by the US government for a range of operating system purchase. Unlike Win32, the POSIX and OS/2 subsystems are optional.

This design simplifies the base operating system and makes it possible to extend the features of an environment subsystem without affecting the Windows NT Executive. Reliability and robustness are improved because applications, subsystems and the executive are protected from each other. This ‘multiple operating system environment’ design approach is similar to that taken by Carnegie Mellon’s Mach operating system, which provides POSIX and 4.3BSD operating system environments as subsystems.



As stated previously, Windows NT uses a *client/server* design in order to support multiple (and fundamentally different) APIs. Environment subsystems implement their APIs, such as Win32, MSDOS, POSIX or OS/2, by calling NT *native services* provided by the NT Executive. Each client application is very tightly coupled to a single environment subsystem, through the API it uses. There is no direct facility for applications to be clients of multiple environment subsystems. These environment subsystems can support multiple client applications.
The NT Executive corresponds to the kernel in a traditional operating system like OS/2 or Unix, except that it only handles low level functions like *memory management*, *process* and *thread creation*, *scheduling*, *interrupt* and *exception* handling, *I/O* and, of course, *object* management. Instead of providing an API, which applications can call, the NT Executive provides a set of object services called *NT native services*, which environment subsystems can call in order to implement their APIs. When an environment subsystem, such as the Win32 subsystem, calls an *NT native service*, the processor traps the call and transfers the call (and any parameters) to the appropriate service routine in the NT executive. The service then runs in kernel mode. The division between user and kernel mode ensures that only specified entry-points (i.e. the NT native services) can be called by environment subsystems.

The *NT Executive* is protected from environment subsystems and client applications because it runs in kernel-mode. Environment subsystems, on the other hand, do not benefit from the same kind of protection because they run in user-mode just like their client applications. That is why they are implemented as protected subsystems and not as Dynamic Link Libraries (DLLs). Environment subsystems are protected because each one runs in a separate process, with a private address space, which is protected from other processes by the NT Executive's Virtual Memory Manager. Client applications cannot call environment subsystem APIs directly, because the environment subsystems cannot share memory with them automatically. Instead, a client application must send a message. The subsystem receives the message, validates the parameters, executes the required API function and returns the results to the client via another message. Thus, a client application never gets direct access to any code or data in the environment subsystem’s address space.

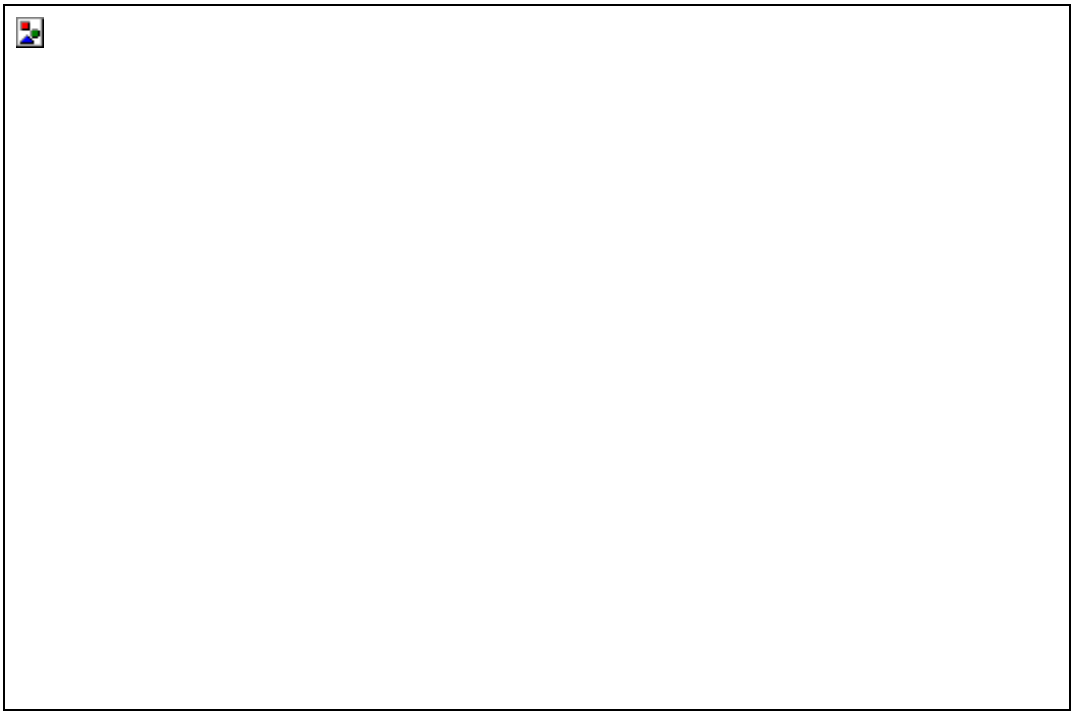


Of all the environment subsystems, the *Win32 subsystem* is the most important. The NT Executive is dependent upon it to provide a user interface and the primary 32-bit programming environment. It manages windows on the screen, displays output on behalf of other environment subsystems, and routes user input to the correct environment subsystem. It also starts applications, irrespective of their type, in response to requests from the Program Manager or from the command line.

The *Win32 subsystem* makes the Win32 API available to applications. This API extends the Windows 3.x API, not only to support the 32-bit flat memory model, but also to add new capabilities. By exporting many of the NT native services it adds a new 'base' API to handle memory management, multitasking, synchronization, I/O, security and exception handling.

Object-based security is implemented in the same way as in the NT Executive. In addition to kernel objects, the Win32 subsystem provides a number of new objects to represent shared objects. These fall into two categories: **user objects** (e.g. windows and menus) and **GDI objects** (e.g. bitmaps and brushes). As in the NT Executive, when an object is created, it is assigned a security descriptor. Subsequent attempts to open a handle to the object are vetted by the Win32 subsystem.

It is reasonably straightforward to port a Windows 3.1 application to Win32. Even if none of the new base and GDI services are used, several immediate benefits occur. For example, all 16-bit Windows applications share the same input queue. Only one application at any one time is processing a message from this queue (i.e. from the keyboard or mouse) - input is strictly serialized and its recipient is decided at input collection time. If an application takes a long time to process a message then it will block all other applications. Each Win32 application thread gets its own input queue and input messages are delivered to this queue at input time. This means that if it a thread is slow in responding to an input message, other application threads are not affected, as the NT kernel will preempt the thread.



A component of the NT Executive, known as the *Local Procedure Call* (LPC) facility, is responsible for passing messages between client applications and environment subsystems. LPC is a variation of *Remote Procedure Call* (RPC), optimized for use between processes on the same computer.

The use of message passing via LPC is transparent to application programmers. As shown in the example above, when a Win32 application calls a Win32 API function, it actually calls a stub function in a Win32 DLL. The stub function packages the API function's parameters into a message, which it sends to the Win32 subsystem. The latter implements the API function and return the results, via a second LPC message, to the DLL. Finally the DLL returns the result to the calling application.

The use of LPC guarantees excellent protection, but it does have a potential impact on performance. In order to minimize this impact, the Windows NT developers at Microsoft developed some specialized forms of LPC. In addition, the developers used some 'tricks' to reduce the frequency of message passing:

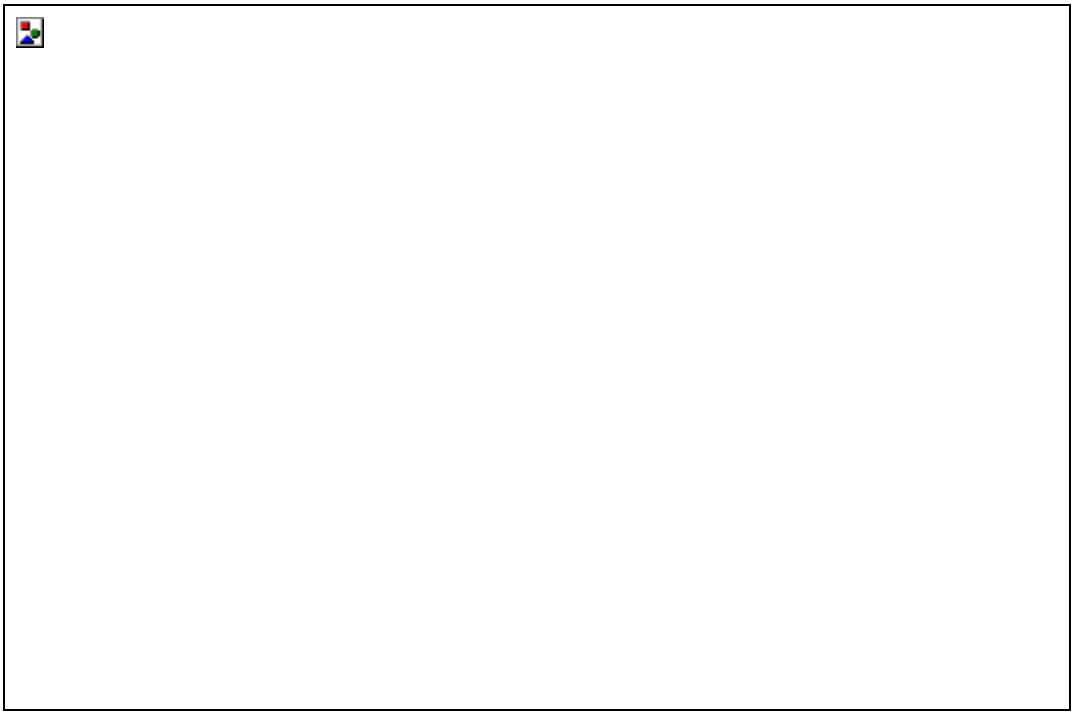
- Using ***client-side*** **DLLs** to implement API routines that don't use or modify global data
- **Storing** certain **subsystem data** in the NT Executive, or caching subsystem data in the client-side DLL
- **Batching client API calls** and sending them to a subsystem in a single message

The primary data structure used to implement the LPC mechanism is the *port object*. The *port object* has an optional shared memory section associated with it to optimize message passing, as described above or if a message becomes too big for pre-defined maximum message size.

The other important feature of this mechanism is the ability of a server to take on the security context of a client by impersonating it, thus ensuring that access to protected objects is carried out in the correct security context.

Windows NT lends itself well to a distributed computing model, since networked computers are based on a client-server model and use messages to communicate. Local servers can easily send messages to remote machines as well. Clients don't need to know whether requests were satisfied locally or remotely.

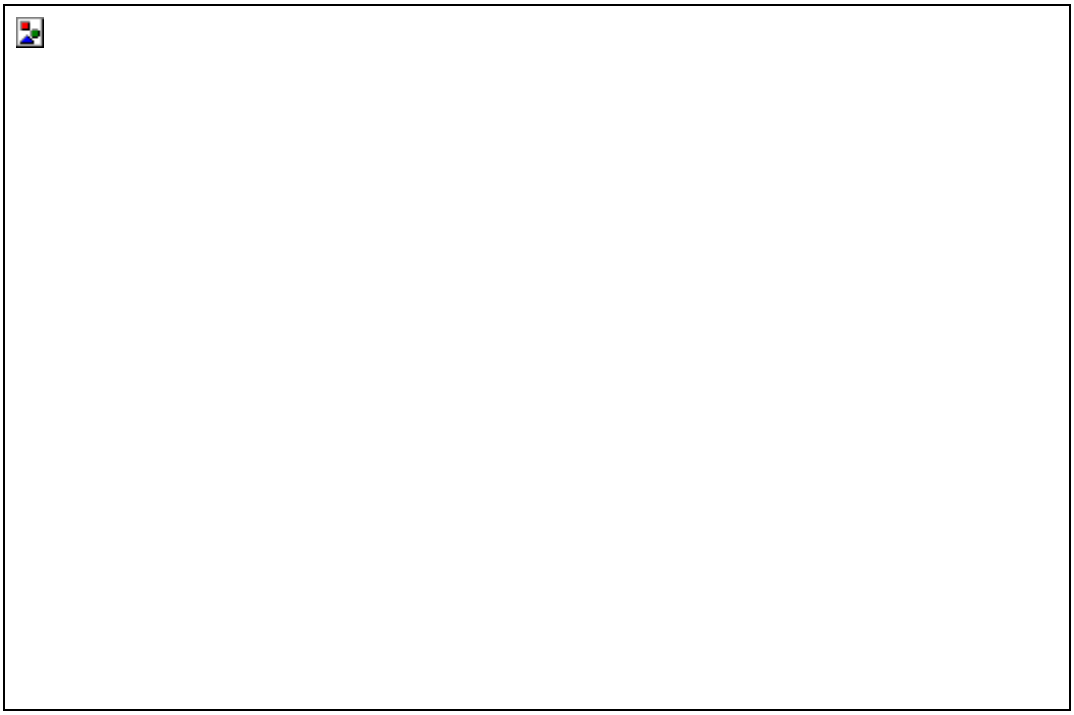
The other mechanism used for client-server communication is the *Event Pair synchronization* mechanism. This is where a dedicated thread in the client and server are able to trade states in a '*set high, wait low*' and '*set low, wait high*' fashion.



More than 98% of Windows NT code, including device drivers, is written in C and C++, which makes it easy to port to other platforms. Windows NT is available on Intel x86 and DEC Alpha platforms, and on multi-processor variants.

Windows NT's portability is also due to other factors:

- The *layered architecture*,
- The *micro-kernel* at the heart of the operating system executive which deals with low-level kernel functionality, and
- The *hardware dependent parts* of the operating system are in a single layer to minimize migration effort.

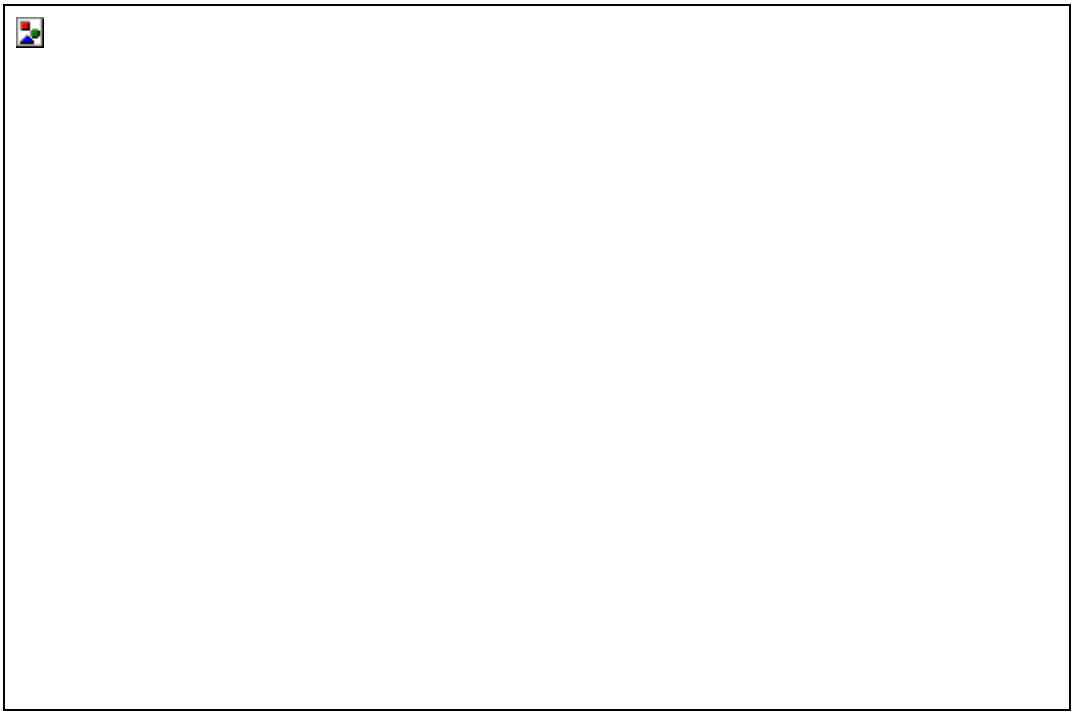


Under Windows NT, each process has its own private, protected address space. The operating system code runs in the most privileged execution mode, and is therefore protected from application and subsystem processes. Applications and sub-systems cannot access the hardware directly; they must go via an API. No process should be able to adversely affect another, or the operating system.

The Windows NT File System provides a transaction-based, fault tolerant file system which logs all file and directory access and can recover very quickly if the system fails unexpectedly. The layered I/O architecture also allows the insertion of other fault tolerant capabilities such as disk mirroring, disk striping and Uninterruptible Power Supply support.

Security is one of the design goals of Windows NT, and is built-in to the system. Windows NT operates an object-based scheme for all system resources. All object manipulation and authentication is handled centrally in a uniform fashion, and thus security validation on the access to objects can be performed in one place; there are no ‘back doors’. Windows NT is C-2 secure (discretionary access control) but has an architecture that will allow it to reach B-level security (mandatory access control).

Windows NT privileges can be assigned to particular users, via password protected logon authentication. Once logged-on, users only have access to the resources for which they have appropriate permission.



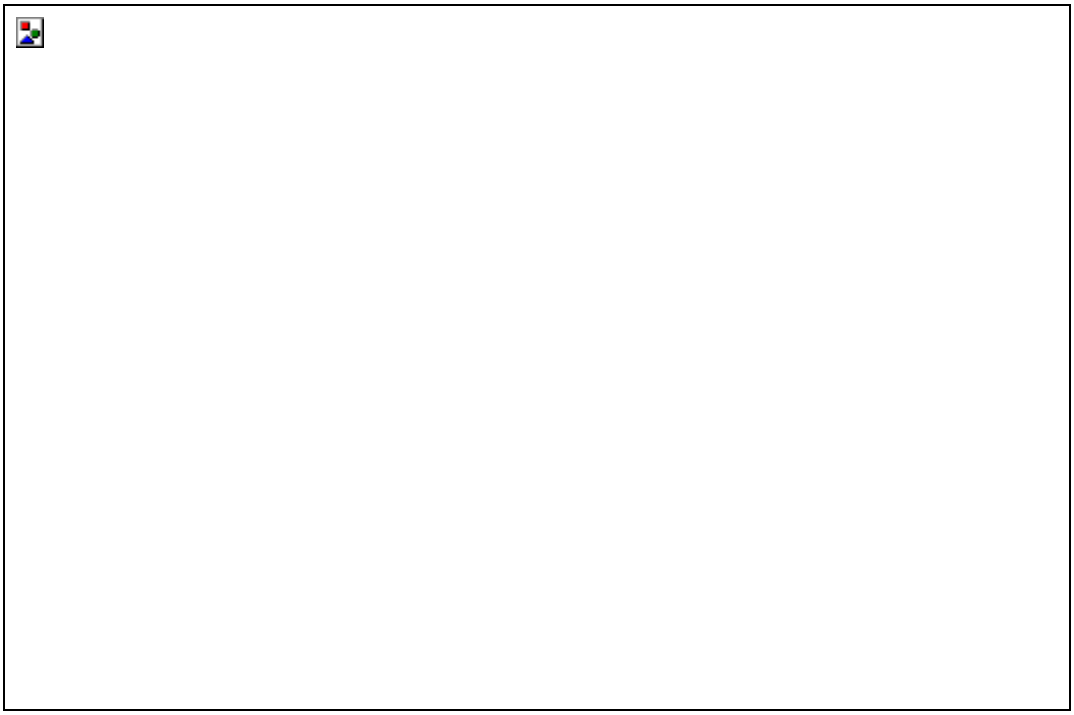
An *object* combines a data structure and the services that operate on that data structure, into a single entity. the basic difference between an object and a data structure is that the object data internals are hidden from view and cannot be accessed directly; the *object's services* (OO methods) provide the only way to access its data or alter its state. This protects the data in the object and also separates the underlying implementation from the code that uses it. An object is referenced by an abstract *handle*, obtained by creating a new object or opening an existing object; rather like the semantics of accessing a file.

Windows NT uses objects to represent many system resources. Shareable executive and kernel resources, including processes, threads, semaphores, open files and shared memory, are implemented as objects and manipulated using object services. Many NT native services are therefore *object services*, i.e. they perform some action on an NT Executive object. Not all data structures in the NT executive are objects. Only **data that needs to be shared, named, protected or made accessible to user mode processes is encapsulated into objects**.

The *Object Manager* in the NT Executive is responsible for creating, deleting, protecting and tracking NT objects, and is the focal point for resource management, e.g. naming resources, placing and enforcing process resource quotas, sharing resources and securing resources against unauthorized access. All NT objects are located in one uniform name space, and are accessed via handles, assigned on a per-process basis. Objects may be unnamed, in which case they can only be referenced by handle, or named, in which case they can also be referenced by name, which is useful for sharing objects between processes.

All objects have a *standard header*, containing attributes and services common to objects of all types, and an object body, containing attributes and services particular to the type of object. Objects of the same type share the same format. When a new object type is registered with the *Object Manager*, a set of methods are registered which are called by the Object Manager at well-defined points in the lifetime of the object. Thus objects are controlled centrally by the *Object Manager* and are manipulated uniformly without regard to type. Security is simplified because all objects are protected and validated in the same way, and object sharing is facilitated via open handles to the object.

The kernel has two kinds of objects; '*dispatcher*' objects have a signal state and control the dispatching and synchronization of system operations, e.g. a thread, and '*control*' objects are used to control the operation of the kernel but do not affect dispatching or synchronization, e.g. a process. Kernel objects are 'wrapped up' by the Object Manager and are used by the Executive to build more complex objects that are, in turn, exported to user-mode for use by the subsystems in their APIs. The subsystems, in turn, provide their own private object representations to their clients. For instance, the Win32 API repackages Executive objects, like native processes and threads, and some new objects, like windows, are defined.



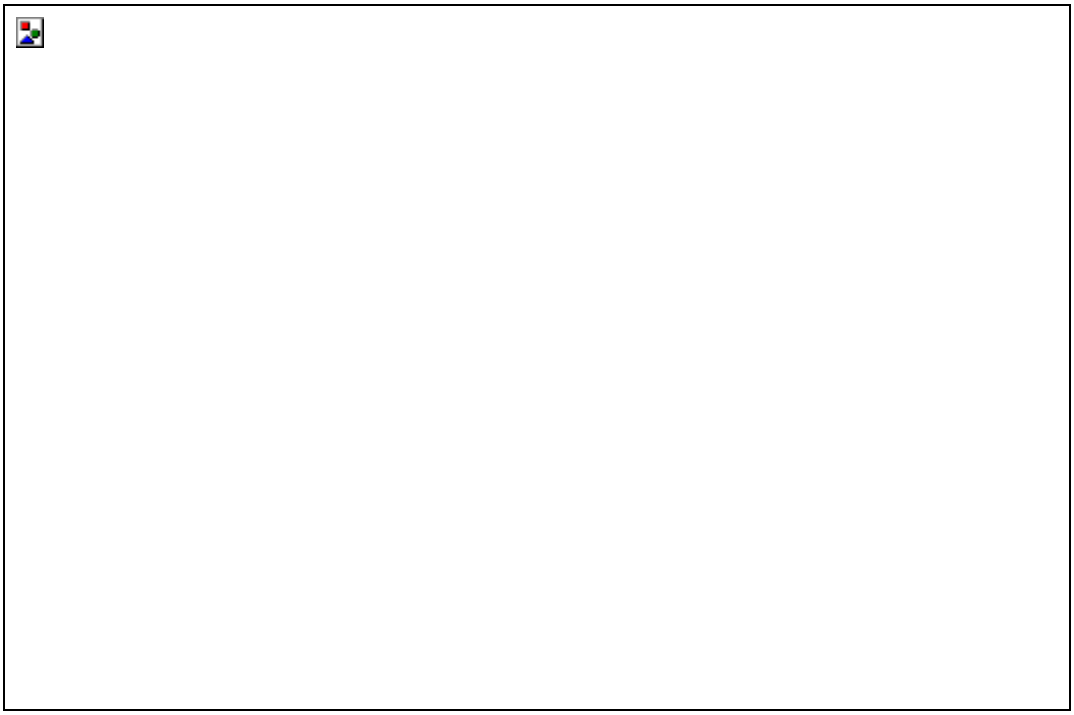
When the NT Executive *Object Manager* is called to create an object it performs a number of functions. After allocating memory for the object, the NT *Object Manager* attaches a *security descriptor* to the object, which specifies who is allowed to use it and what they are allowed to do with it. It also creates an entry in an object directory structure in which all object names are stored. Finally it creates an *object handle*, which is returned to the caller.

All user-mode processes, including environment subsystems, must have a handle to an object before they can access the object. A process can acquire an object handle in any one of the following ways:

- Creating a new object, named or unnamed
- Opening a handle to an existing named object
- Inheriting a handle from another process
- Receiving a duplicated handle from another process

Object *names are visible across process boundaries*, so naming an object allows other processes to 'open' an existing object if it knows how to name it. For unnamed objects, *open handles can be 'inherited'* by related processes, or can be passed to unrelated processes to be *'duplicated'*.

An open object handle is invalid when it is closed, and an object is removed from the system when all open handles referencing it are closed. All open object handles are closed when a process exits.
Except for what they refer to, there is no difference between handles to different types of object. The NT Executive maintains, for each process, a single object table, which contains pointers to all objects that the process has a handle to. An NT object handle is actually an index into this table. Each entry in the table also contains the corresponding handle's granted access rights and a flag, which specifies whether a child processes, will inherit the handle.
Two processes share an object when they both have open handles to it. The NT *Object Manager* monitors who is using an object and will not delete an object while any object has a handle to it. The NT *Object Manager* also monitors a process's usage of resources. Each time a process opens an object handle, the NT Object Manager charges the process for the memory the object uses. Therefore, it will not allow a process to open an object if, in so doing, the total charge would exceed the quota a system administrator has set on the user represented by the process.

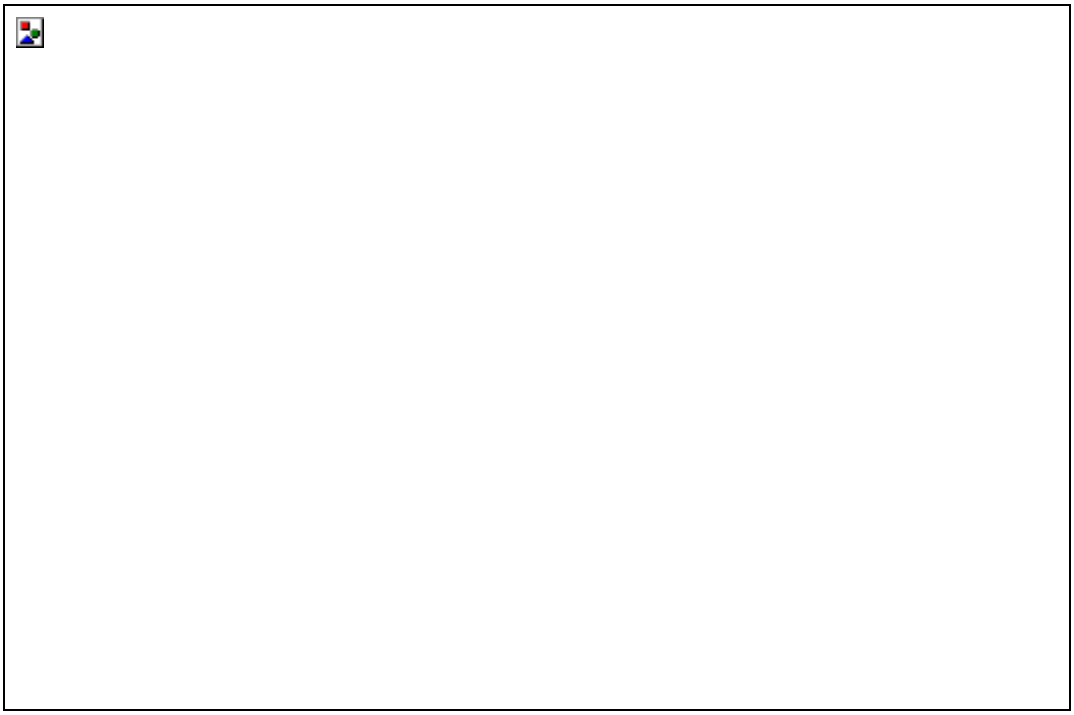


Windows NT is designed to meet the *Class C2 level of security* defined by the U.S. Department of Defense. Basically this means that the creator of a system resource provides discretionary access control to that resource and has the right to decide who can access it and what they can do with it. The operating system should also detect and record attempts to create, access or delete system resources. The use of objects in Windows NT simplifies security. When a user process attempts to access any system resource, the NT *Object Manager* performs a *security check* to ensure that the process has the necessary access rights.

In order to identify users, every user of Windows NT is required to log on. The user-mode Logon process prompts the user for an account name and password. This information is then passed to the *Security Subsystem*, which checks it against a security database. If the logon is authentic, the *Security Subsystem* creates an object, called an *access token*, which is permanently attached to any process representing that user. The key attribute in the access token is a security ID that normally corresponds to the user's logon account name.

All objects that can be shared between two or more processes are assigned *security descriptors* when they are created. **A security descriptor contains an *Access Control List* (ACL) of *Access Control Entries* (ACEs). Each ACE contains a security ID and a set of access rights**. When a user process attempts to open a handle to an object, specifying its desired access rights, the Object Manager calls the *Security Reference Monitor*. The Security Reference Monitor checks the access token of the user process against the ACL to determine whether the process can access the object. If the Security Reference Monitor finds an ACE that allows access, it returns a handle that contains the granted access rights. On subsequent calls to the API that make use of this handle, the Object Manager simply compares the granted access rights with the type of access implied in the call.

A Win32 process supplies a security attribute when creating a securable object. This defines a security descriptor, which dictates how the object can be accessed, and a flag defining whether the object handle can be inherited for use by related processes. If no security attribute is specified, the handle cannot be inherited and no security checks are enforced on API access to the object. If the security attribute is specified, but the security descriptor is not, then the object gets the default security descriptor for the object type. One process spawning another, can also decide whether it will allow the new process to inherit any, or all, of the inheritable object handles it can see. Inherited open handles have the same access as the parent's handle.

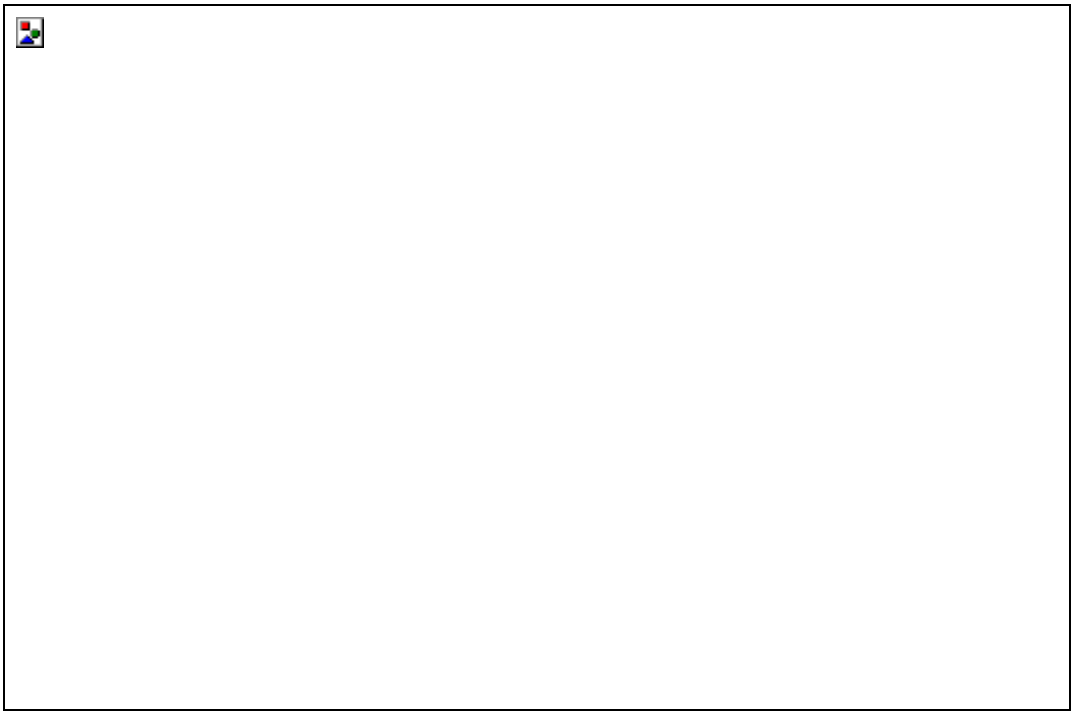


Windows NT offers pre-emptive multi-tasking. Processes can divide their workload amongst simultaneous threads of execution. That is to say, a program can execute in two or more places at the same time. NT preemptively schedules these threads on a priority basis; the system always runs the thread with the highest priority. Foreground user interface threads are given a priority 'boost', as are some key system events. Important time-critical tasks, like data capture, can be given a higher priority via a programming API. So, the system always appears responsive.

To appreciate this type of multi-tasking, contrast it with the non-preemptive multi-tasking of Windows 3.x. Windows 3.x applications are event-driven by processing messages which are generated by the system and by applications. In this scenario, applications are allocated CPU time on a per-message basis. Message processing is sequential; so only one application can process one message at any given time. If one application decides to take a substantial amount of time to process one message, then all other applications are locked out, and the system becomes unresponsive. The reason this serialization is imposed is to ensure a particular 'input-ahead' event model to cope with the user generating input faster than applications can process it. User input is buffered up in a single place by the system and is not directed to a particular application until it is to be processed.

Win32 applications, which create graphical windows, also process messages, but the message model has been de-synchronized. A different 'input-ahead' model has been chosen. User input is buffered on a per-thread basis at the time it is generated. This means that one thread, which takes a long time to process a message, will not adversely affect any other threads; it will of course affect the windows of that thread. The user can switch away from a thread performing lengthy processing, and access other applications at any time.

Windows NT also supports symmetric multi-processing hardware. Symmetric multi-processing means that all processors are considered peers; they are all of the same type and have the same view of the system, and are capable of performing the same work. The operating system design doesn't dedicate any processor to particular tasks, for example I/O; NT can dispatch any thread to run on any processor. This eliminates 'bottlenecks' in the system, and permits scaleable system performance. Adding more processors, within reason, will make the system run linearly faster. Applications and sub-system run unchanged on multi-processor systems because all multi-processing logic is embedded in the kernel.



The *Kernel* component of the NT *Executive* is responsible for (a) determining how the rest of the operating system, and applications, use the processor (or processors in multi-processor systems), and (b) for optimizing the utilization of the processor(s).

The *Kernel* schedules threads for execution. A *thread* is a unit of execution that runs in the *virtual address space* of a process, using resources allocated to that process. The Kernel maintains a list of threads that are ready to execute, but are simply waiting their turn. When a currently executing thread reaches the end of its time slice, or quantum (about 20 ins.), the kernel selects the highest priority thread from the ready list and performs a context switch to it. A context switch involves suspending the execution of the current thread, saving its state (e.g. the contents of the processor registers), restoring the state of the new thread and allowing the new thread to resume execution. In most cases, a thread will not run for its complete time quantum, either because it blocks on an API function, or because a higher-priority thread becomes ready. With the exception of the Kernel component, any thread can be preempted at any time.

The *Kernel* also handles hardware interrupts and processor exceptions. A processor raises an exception when it detects an error or particular condition while executing instructions. Examples of exceptions include general protection, page fault, divide-by-zero and debug instructions. Note that all interrupt and exception handling occurs in kernel mode. (In Windows NT it is not possible for any application to perform I/O or handle interrupts directly.)

Windows NT supports *symmetric multiprocessing* as well as multitasking. Multiprocessing is the ability to execute multiple threads on multiple processors. Symmetric multiprocessing means that any thread can execute on any processor. Thus, on a multiprocessor system with eight processors, eight threads can execute simultaneously! The Kernel component is responsible for synchronizing the processors.