C H A P T E R   4

Robustness Improvements

Windows 95 improves on the robustness of Windows 3.1 to provide great support for running MS-DOS–, Win16–, and Win32–based applications, and provides a high level of system protection from errant applications.

Windows 3.1 provided a number of mechanisms to support a more robust and stable environment over Windows 3.0. These improvements included:

u **Better resource cleanup.** When a Windows or MS-DOS–based application crashed, users were able to continue running such that they could save their work.

u **Local reboot.** This allowed users to shut down an application that hung.

u **Parameter validation for API calls.** This allowed the system to catch many common application errors and fail the API call, rather than allowing bad data to be passed to an API.

While the work done in Windows 3.1 provided a more robust and stable environment than Windows 3.0, we made it even better in Windows 95.

System-wide Robustness Improvements

System-wide improvements resulting in a more robust operating system environment than Windows 3.1 include:

u Better local reboot

u Virtual device driver (VxD) thread cleanup when a process ends

u Per-thread state tracking

u Virtual device driver parameter validation

Filename: in.doc   Project: *Insert existing text here and delete this text. Do not remove the following paragraph.
Template: Author: Shane A. Gonzales   Last Saved By: Shane A. Gonzales
Revision #: 2   Page: 79 of 12   Printed: !Unexpected End of Expression

Better Local Reboot

The ability for a user to end an application or a virtual machine (VM) that hangs is called a *local reboot*. With Windows 3.1, users were able to perform a local reboot for an application or VM that the system thought was hung by pressing the three-key Ctrl-Alt-Del combination. Users could pretty easily end errant VMs with the local reboot request, however requesting a local reboot for a Windows–based application often resulted in bringing the entire system down or not allowing the user to end the errant Windows–based process.

Windows 95 greatly improves upon the local reboot support by providing a means to end an MS-DOS–based application running in a VM, end a Win16–based application, or end a Win32–based application, in a manner without bringing down the entire system. The process of cleaning up the system after a local reboot is now more complete than for Windows 3.1. This process is described more fully later in this chapter.

When a user requests a local reboot, the Windows 95 system displays the Close Program dialog box identifying the different tasks that are running and the state that the system perceives each to be in. This level of detail affords the user much more flexibility and control over local reboot than with Windows 3.1.
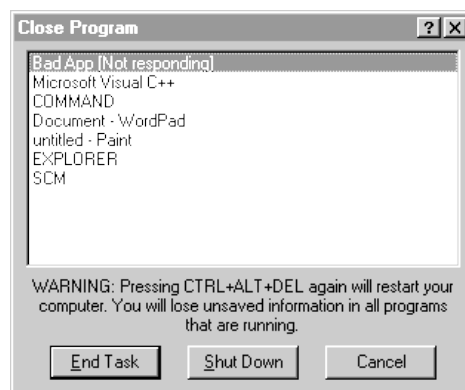

Figure 1. Close Program Dialog Box in Windows 95

Applications are identified as "not responding" when they haven't checked the message queue for a period of time. Although an application may be performing a computationally-intensive operation, a well-behaved application will check the message queue on a more frequent basis. Just as with Windows 3.1, it is necessary for a Win16–based application to check the message queue in order to relinquish control to other tasks running.

Filename: in.doc   Project: *Insert existing text here and delete this text. Do not remove the following paragraph.
Template: Author: Shane A. Gonzales   Last Saved By: Shane A. Gonzales
Revision #: 2   Page: 80 of 12   Printed: !Unexpected End of Expression

Virtual Device Driver Thread Clean-up When a Process Ends

Local reboot support is also aided by improved VxD thread clean-up when a given process ends. With Windows 3.1, it was quite common for the system to be unable to recover if the system was running real-mode code such as BIOS routines when an application ended abnormally, or if the user requested a local reboot to end a seemingly-hung application. For example, suppose the user requested a local reboot or suppose an operation (such as a network operation in real-mode, a disk I/O, or an asynchronous application request) ended abnormally because of another application-based error. In these cases, Windows 3.1 couldn't necessarily clean up properly to free allocated resources, and possibly couldn't even return control to the user.

Windows 95 improves system clean-up by providing each system VxD the ability to track the resources it allocates on a per-thread basis. Since most computer system functionality and support is handled by VxDs in Windows 95 rather than by real-mode code or BIOS routines, Windows 95 can recover from errors or situations that, under Windows 3.1, would required the computer to be rebooted.

When Windows 95 ends a given thread, each VxD receives notification that the thread is ended (because the user exited the application, a local reboot was requested, or the application ended abnormally). This notification allows the VxD to safely cancel any operations it is waiting to finish. This also frees any resources that the VxD previously allocated for the thread or application. Since the system tracks an entire VM, a Win16 application, and a Win32 thread, each as a per-thread instance, the system can clean up properly at each of these levels, without affecting the integrity of the system.

Per-Thread State Tracking

Resource tracking in Windows 95 is much better than that provided in Windows 3.1 to aid system clean-up. In addition to tracking resources on a per-thread basis by system VxDs, resources such as memory blocks, memory handles, graphics objects, and other system items are allocated and also tracked by system components on a per-thread basis. Tracking these resources on a per-thread basis allows the system to clean up safely when a given thread ends, either normally at the user's request, or abnormally. Resources are identified and tracked by both a thread ID, and by the major version number of Windows that is stored in the .EXE header of the application.

For a discussion of how the thread ID and the version number of Windows are used to facilitate cleanup of the system and recovery of allocated resources for Win16 and Win32–based applications, see the Win16 and Win32–based application robustness sections in this guide.

Filename: in.doc   Project: *Insert existing text here and delete this text. Do not remove the following paragraph.
Template: Author: Shane A. Gonzales   Last Saved By: Shane A. Gonzales
Revision #: 2   Page: 81 of 12   Printed: !Unexpected End of Expression

Virtual Device Driver Parameter Validation

Virtual device drivers are an integral part of the Windows 95 operating system and have a more important role than in Windows 3.1, as many operating system components are implemented as VxDs. To help provide for a more stable and reliable operating system, Windows 95 provides support for parameter validation of virtual device drivers, something that was not available for Windows 3.1. The debug version of system files for Windows 95 provided as part of the SDK for Windows 95 and DDK for Windows 95 will aid VxD developers to debug their VxDs during the course of development to ensure their VxDs are stable and robust.

In addition to providing improved system-wide robustness, Windows 95 delivers improved robustness for running MS-DOS–based, Win16–based, and Win32–based applications, providing for a more stable and reliable environment than Windows 3.1.

Robustness for MS-DOS–based Applications

Windows 95 provides improved support for running MS-DOS–based applications under Windows 95 that were not possible with Windows 3.1. Several improvements present in Windows 95 provide great robustness for running MS-DOS–based applications. These improvements are described in the next two sections.

Improved Protection for Virtual Machines

Each MS-DOS–based application runs in a separate VM, and are configured by default to execute preemptively and run in the background when another application is active. Each VM is protected from other tasks running in the system, and an errant Win16– or Win32–based application can't crash a running MS-DOS–based application, and vice versa.

Under Windows 3.1, each VM inherits the attributes and environment configuration from the global System VM. While each VM is protected from another VM preventing errant MS-DOS–based applications from accessing memory or overwriting system code thus possibly bringing the system down, the VM does not provide complete protection preventing an MS-DOS–based application from overwriting MS-DOS system code. MS-DOS–based applications have full access to all memory locations in the first megabyte of addressable memory space (i.e., the real-mode memory range).

Filename: in.doc   Project: *Insert existing text here and delete this text. Do not remove the following paragraph.
Template: Author: Shane A. Gonzales   Last Saved By: Shane A. Gonzales
Revision #: 2   Page: 82 of 12   Printed: !Unexpected End of Expression

Windows 95 supports a higher level of memory protection for running MS-DOS–based applications, preventing the applications from overwriting the MS-DOS system area in real-mode. Users can configure their MS-DOS–based applications to run with "general memory protection" enabled if they want the highest level of system protection. This mode is not enabled by default due to overhead required to validate memory access requests. Furthermore, parameter validation of Int 21h operations on pointers will be performed. This will increase the robustness of the system.

Better Cleanup When a Virtual Machine Ends

When a VM ends in Windows 95—either normally because the exited the application or VM or requested a local reboot, or abnormally because the application ends abnormally—the system frees all resources allocated for the VM. In addition to the resources allocated and maintained by the system VxDs as previously discussed, the system tracks resources allocated for the VM by the Virtual Machine Manager, including DPMI and XMS memory that the VM requested.

In Windows 3.1, resources such as DPMI memory are not released properly when the VM is ended. Windows 95 frees the DPMI memory used by the VM and other resources allocated by the operating system components.

Robustness for Win16–based Applications

Windows 95 provides improved support for running Win16–based applications. It also provides great robust Win16 application support plus compatibility with existing Windows–based applications, while keeping the memory requirements low. The next two sections describe improvements for Win16–based applications running under Windows 95.

Filename: in.doc   Project: *Insert existing text here and delete this text. Do not remove the following paragraph.
Template: Author: Shane A. Gonzales   Last Saved By: Shane A. Gonzales
Revision #: 2   Page: 83 of 12   Printed: !Unexpected End of Expression

Per-Thread State Tracking

Under Windows 3.1, when a Windows–based application ended, the resources used by the application were not released by the system. Some Windows–based applications took this into account and didn't free certain resources as the allocated resources could then be accessed by other in-memory Windows–based applications or system components (such as DLLs). Changing the way the system behaves when a Win16 application ended—for example, by freeing up all resources allocated to the Win16 application immediately—might break an existing application.

Under Windows 95, each Win16–based application runs as a separate thread in the Win16 address space to facilitate resource tracking. When a Win16 application ends, resources allocated to the Win16 application aren't immediately released by the system but are held by the system until the system can safely free them. When the last Windows 3.x application is ended, Windows 95 determines that it is safe to free all resources allocated for Win16–based applications and begins cleaning the system of resources associated with Windows 3.x applications. Windows 95 determines that no more Win16–based applications are running by associating the Windows version number of the application with the thread ID for the running process. When no more Windows 3.x applications are running in the system, Windows 95 frees any remaining resources allocated by the Win16–based applications.

Filename: in.doc   Project: *Insert existing text here and delete this text. Do not remove the following paragraph.
Template: Author: Shane A. Gonzales   Last Saved By: Shane A. Gonzales
Revision #: 2   Page: 84 of 12   Printed: !Unexpected End of Expression

Parameter Validation for Win16 APIs

Windows 95 provides support for checking the validity of parameters passed to Windows APIs by Win16–based applications. Some users perceived Windows 3.0 to be unstable because the "Unrecoverable Application Errors" (UAE) were common when working with Windows–based applications. Most of this instability was in fact caused by Windows–based applications that passed invalid parameters to Windows API functions. The APIs in turn attempted to process this bad data and usually attempted to access an invalid area of memory. For example, when an application that inadvertently passed a NULL pointer to a Windows API function which tried to access memory at the address referenced, it would generate a UAE or "general protection fault."

Windows 95 provides parameter validations for all Win16–based APIs and checks incoming data to API functions to ensure the data is valid. For example, functions that reference memory are checked for NULL pointers, and functions that operate on data within a range of values are checked to ensure the data is within the proper range. If invalid data is found, an appropriate error number will be returned to the application. It is then up to the application to catch the error condition and handle it accordingly.

The SDK for Windows 95 provides debug system components to aid software developers to debug their applications. The debug components for Windows 95 provide extensive error reporting for parameter validation to aid the developer in tracking common problems related to invalid parameters during the course of development.

Filename: in.doc   Project: *Insert existing text here and delete this text. Do not remove the following paragraph.
Template: Author: Shane A. Gonzales   Last Saved By: Shane A. Gonzales
Revision #: 2   Page: 85 of 12   Printed: !Unexpected End of Expression

Robustness for Win32–based Applications

While the robustness improvements for running MS-DOS–based and Win16–based applications in Windows 95 is better than that provided by Windows 3.1, the greatest support for robustness in Windows 95 is available when running Win32–based applications. Win32–based applications also benefit from preemptive multitasking, linear address space (rather than segmented), and support for a feature-rich API set.

Robustness support for Win32–based applications includes:

u   A private address space for each Win32–based application to run, segregating and protecting one application from others that are running concurrently

u   Win32 APIs that support parameter validation to provide for a stable and reliable environment

u   Resources are tracked by threads and freed up immediately when the thread ends

u   Separate message queues are used to ensure that a hung Win32–based application will not suspend the entire system

Each Win32–based Application Runs in its own Private Address Space

Each Win32–based application runs in its own private address space. This provides protection of its resources at the system level from other applications running in the system. It also prevents other applications inadvertently overwriting the memory area of a given Win32–based application, and prevents the Win32–based application from inadvertently overwriting the memory area of another application or the system as a whole.

Parameter Validation for Win32 APIs

As with parameter validation for Win16–based applications, Windows 95 provides parameter validation for Win32 APIs used by Win32–based applications. The SDK for Windows 95 helps software developers debug errors resulting from attempts to pass invalid parameters to Windows APIs. For additional information about parameter validation for Win16 APIs, see the discussion of robustness for Win16–based applications presented earlier in this guide.

Filename: in.doc   Project: *Insert existing text here and delete this text. Do not remove the following paragraph.
Template: Author: Shane A. Gonzales   Last Saved By: Shane A. Gonzales
Revision #: 2   Page: 86 of 12   Printed: !Unexpected End of Expression

Per-Thread Resource Tracking

Resources allocated by threads in Win32–based applications are tracked by the system. Unlike thread tracking for Win16–based applications, any allocated Win32 resources are automatically deallocated when the thread ends processing. This helps to ensure that allocation system resources are freed immediately and are available for use by other running tasks.

Resources are cleaned up properly when threads either end execution on their own (for example, perhaps the developer inadvertently did not free allocated resources), or when the user requests a local reboot that ends a given Win32 application thread or process. Unlike Win16–based applications designed to run under Windows 3.1, Win32–based applications free up allocated resources immediately when the application or a separate thread ends.

Separate Message Queues for Win32–based Applications

The Windows environment performs tasks based on the receipt of messages sent by system components. Each message is generated based on an action or *event* that occurs on the system. For example, when a user presses a key on the keyboard and releases it, or moves the mouse, a message is generated by the system and passed to the active application informing it of the event that occurred. Windows–based applications call specific Windows API functions to extract event messages from message queues and perform operations on the messages (for example, accept an incoming character typed on the keyboard, or move the mouse cursor to another place on the screen).

Under Windows 3.1, a single message queue was used by the entire system. Win16–based applications cooperatively examined the queue and extracted messages destined to them. This single-queue scheme posed some problems. For example, if a Win16–based application hung and prevented other applications from checking the message queue, the message queue became full and accepted no new messages. Then other Win16–based applications were suspended until control was relinquished to them and they were able to check for event messages.

Filename: in.doc   Project: *Insert existing text here and delete this text. Do not remove the following paragraph.
Template: Author: Shane A. Gonzales   Last Saved By: Shane A. Gonzales
Revision #: 2   Page: 87 of 12   Printed: !Unexpected End of Expression

Windows 95 solves the problems inherent with a single message queue in Windows 3.1, by providing for separate message queues for each running Win32–based application (see Figure 2). The system takes messages from the input message queue and passes them to the message to the appropriate Win32–based application or to the Win16 Subsystem, if the message is destined for a Win16–based application. If a Win32–based application hangs and no longer accepts and processes incoming messages destined for it, the Win32–based application does not affect other Win16– and Win32–based applications currently running.
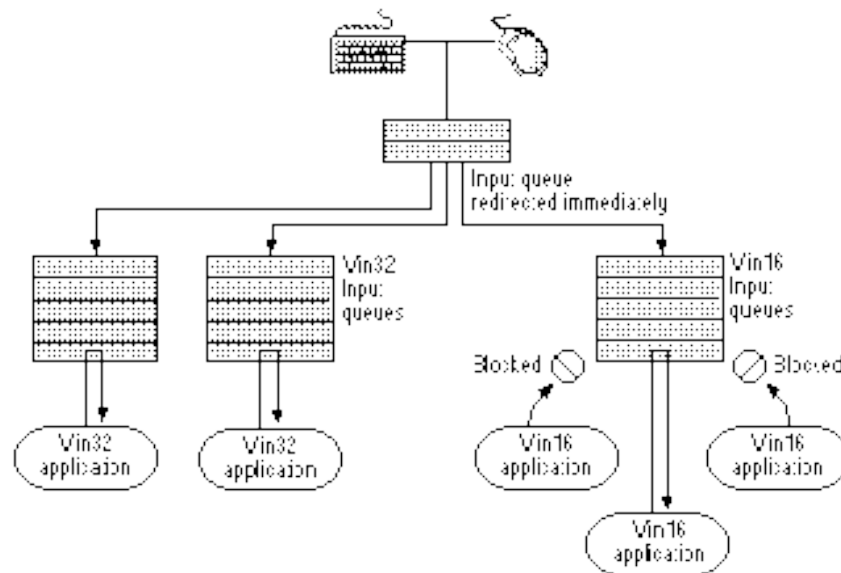
Figure 2. Win32–based Applications Use Separate Message Queues for Increased Robustness

If a Win32–based application ends or the user requests a local reboot operation on a Win32–based application, having separate message queues improves the robustness of the operating system by making it easier to clean up and to free system resources used by the application. It also provides greater reliability and recoverability if an application hangs.

Filename: in.doc   Project: *Insert existing text here and delete this text. Do not remove the following paragraph.
Template: Author: Shane A. Gonzales   Last Saved By: Shane A. Gonzales
Revision #: 2   Page: 88 of 12   Printed: !Unexpected End of Expression

Improved Local Reboot Effectiveness

Due to the robustness improvements supported in the system for Win32–based applications (including the use of a private address space, separate message queues, and resource tracking by thread), users should be able to end ill-behaved Win32–based applications in almost all cases, without affecting the integrity of the Windows system or other running applications.

When a Win32–based application is ended, resources are deallocated and cleaned up by the system as soon as the application ends. Because Win32–based applications run in their individually-allocated environment, this method is even more robust than the way Windows 95 is able to reallocate Win16 application resources. (See the section called "Robustness for Win16-based Applications" for more details.)

Structured Exception Handling

An *exception* is an event that occurs during the execution of a program, and that requires the execution of software outside the normal flow of control. Hardware exceptions can result from the execution if certain instruction sequences, such as division by zero or an attempt to access an invalid memory address. A software routine can also initiate an exception explicitly.

The Microsoft Win32 application programming interface (API) supports *structured exception handling*, a mechanism for handling hardware- and software-generated exceptions. Structured exception handling gives programmers complete control over the handling of exceptions. The Win32 API also supports termination handling, which enables programmers to ensure that whenever a guarded body of code is executed, a specific block of termination code is also executed. The termination code is execute regardless of how the flow of control leaves the guarded body. For example, a termination handler can guarantee that clean-up tasks are performed even if an exception or some other error occurs while the guarded body of code is being executed. Structured exception and termination handling is an integral part of the Win32 system and it enables a very robust implementation of system software.

Windows 95 provides structured exception and termination handling for Win32–based applications that make use of this functionality—resulting in applications that can identify and rectify error conditions that may occur outside their realm of control, providing a more robust computing environment.

Filename: in.doc   Project: *Insert existing text here and delete this text. Do not remove the following paragraph.
Template: Author: Shane A. Gonzales   Last Saved By: Shane A. Gonzales
Revision #: 2   Page: 89 of 12   Printed: !Unexpected End of Expression

Try It!

Mouse

To see how the robustness improvements made in Windows 95 results in a more stable, and reliable environment than Windows 3.1, you've got to try it!

Local Reboot

To see how local reboot works in Windows 95 and Windows 3.1, you've got to try the three-finger salute. With a couple of applications running in the system, press CTRL-ALT-DEL simultaneously.

Under Windows 3.1, the system may identify the currently active application as the application that has the focus of the local reboot request, or may report back that there is no application in a hung or inactive state.

Under Windows 95, the user will be presented with a list of active applications, and is given the option of terminating any currently running tasks. Applications that are no longer responding to the system are identified as being "not responding" in the local reboot dialog box.

Filename: in.doc   Project: *Insert existing text here and delete this text. Do not remove the following paragraph.
Template: Author: Shane A. Gonzales   Last Saved By: Shane A. Gonzales
Revision #: 2   Page: 90 of 12   Printed: !Unexpected End of Expression