CHAPTER 5

Robustness

Windows 95 improves on the robustness of Windows 3.1 to provide great support for running MS-DOS–based, Win16–based, and Win32–based applications, and to provide 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, including the following:

- **Better resource clean-up.** When an MS-DOS–based or Windows–based application crashed, users could continue running in a way that allowed them to save their work.
- **Local reboot.** Users could shut down an application that hung.
- **Parameter validation for API calls.** The system could catch many common application errors and fail the API call, rather than allow bad data to be passed to an API.

Just as the improvements in Windows 3.1 provided a more robust and stable environment than Windows 3.0, the improvements in Windows 95 provide an even better environment.

System-Wide Robustness Improvements

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

- Better local reboot
- Virtual device driver thread clean-up when a process ends
- Per-thread state tracking
- Virtual device driver parameter validation

Better Local Reboot

The capability whereby users can end an application or VM that hangs is called a *local reboot*. With Windows 3.1, users could perform a local reboot by pressing the three-key CTRL+ALT+DEL combination. Users could pretty easily end errant VMs with a local reboot request, but a local reboot request for a Windows–based application often didn't end the errant Windows–based process or brought the entire system down.

Windows 95 greatly improves the local reboot support by providing a means to end an MS-DOS–based application running in a VM, a Win16–based application, or a Win32–

based application without bringing down the entire system. Moreover, the process of cleaning up the system after a local reboot is now more complete than for Windows 3.1. (This process is described later in this chapter.)

In Windows 3.1, when a user requests a local reboot, the system may identify the active application as the application that has the focus of the local reboot request, or it may report back that there is no application in a hung or inactive state. In Windows 95, the system displays the Close Program dialog box, which identifies the tasks that are running and the state that the system perceives each one to be in, as shown in Figure 37. This level of detail affords the user much more flexibility and control over the local reboot than with Windows 3.1.

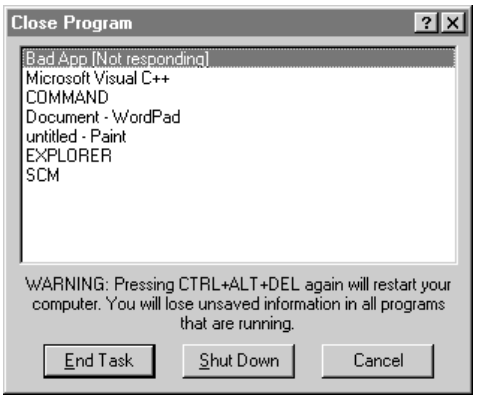

Figure 37. The Close Program dialog box

Applications are identified as "not responding" when they haven't checked the message queue for a period of time. Although some applications don't check the message queue while performing computationally intensive operations, well-behaved applications check the message queue frequently. In Windows 95, as in Windows 3.1, Win16–based applications must check the message queue to relinquish control to other running tasks.

Try It!

Perform a Local Reboot

1. With a couple of applications running, press CTRL+ALT+DEL. You are presented with a list of active applications. Applications that are hung are identified as *Not responding*.

2. Terminate one of the tasks by clicking *End Task*.

Virtual Device Driver Thread Clean-Up

Local reboot support is also aided by improved VxD thread clean-up when a given process ends. With Windows 3.1, the system often couldn't recover either if it 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, if 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, Windows 3.1 sometimes couldn't clean up properly to free allocated resources and sometimes couldn't even return control to the user.

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

When Windows 95 ends a given thread (because the user exited the application, a local reboot was requested, or the application ended abnormally), each VxD receives notification that the thread is ended. This notification allows the VxD to safely cancel any operations it is waiting to finish and frees any resources that the VxD previously allocated for the thread or application. Because the system tracks each VM, Win16–based application, and Win32 thread as a separate 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

To aid in system clean-up, resource tracking in Windows 95 is much better than in Windows 3.1. 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 the major Windows version number that is stored in the .EXE header of the application.

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

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, because many operating system components are implemented as VxDs. To help provide a more stable and reliable operating system, Windows 95 provides support for parameter validation of virtual device drivers, which was not available for Windows 3.1. The debug version of Windows 95 system files provided as part of the Windows 95 SDK and Windows 95 DDK can be used by VxD developers to debug their VxDs during the course of development, ensuring that their VxDs are stable and robust.

In addition to improving system-wide robustness, Windows 95 provides improved robustness for running MS-DOS–based, Win16–based, and Win32–based applications, which also ensures that Windows 95 is a more stable and reliable environment than Windows 3.1.

Robustness for MS-DOS–Based Applications

Because of improved support, users can run MS-DOS–based applications under Windows 95 that they could not run under Windows 3.1. Several improvements that provide great robustness for running MS-DOS–based applications are described in the next two sections.

Virtual Machines Protection Improvements

Each MS-DOS–based application runs in a separate VM and is 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, so an errant Win16–based or Win32–based application can't crash a running MS-DOS–based application, and vice versa.

Under Windows 3.1, each VM inherited the attributes and environment configuration from the global System VM. Each VM was protected from other VMs, preventing errant MS-DOS–based applications from accessing memory or overwriting system code and thus possibly bringing the system down. However, the VMs did not completely prevent an MS-DOS–based application from overwriting MS-DOS system code, because MS-DOS–based applications had full access to all memory locations in the first megabyte of addressable memory space (the real-mode memory range).

Windows 95 provides a higher level of memory protection for running MS-DOS–based applications by preventing the applications from overwriting the MS-DOS system area in real mode. If users want the highest level of system protection, they can configure their MS-DOS–based applications to run with general memory protection enabled. (This mode is not enabled by default because of the overhead required to validate memory access requests.) In addition, parameter validation of Int 21h operations on pointers is performed, thereby increasing the robustness of the system.

Better Clean-Up When a Virtual Machine Ends

When a VM ends in Windows 3.1, some resources, such as DPMI memory, are not released properly. When a VM ends in Windows 95—either normally because the user exited the application or VM or requested a local reboot, or abnormally because the application hung—the system frees all resources allocated to the VM. These resources include not only those allocated and maintained by the system VxDs, but also those allocated for the VM by the Virtual Machine Manager, including any DPMI and XMS memory that the VM requested.

Robustness for Win16–Based Applications

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

Per-Thread State Tracking

With Windows 3.1, when a Windows–based application ended, the resources that had been used by the application were not released by the system. Some Windows–based applications took this behavior into account and didn't free certain resources, so that their allocated resources could be accessed by other in-memory Windows–based applications or by system components such as DLLs.

Changing the way the system behaves when a Win16–based application ends—for example, immediately freeing up all the resources allocated to the application—might have resulted in the breaking of existing applications. To facilitate resource tracking under Windows 95, each Win16–based application runs as a separate thread in the Win16 address space. When a Win16–based application ends, Windows 95 doesn't immediately release the resources allocated to the application but holds them until the last Win16–based application has ended. (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 the last Win16–based application has ended and it is safe to free all resources allocated to Win16–based applications, Windows 95 begins releasing the resources.

Parameter Validation for Win16 APIs

Windows 3.0 was perceived by some users as unstable because Unrecoverable Application Errors (UAEs) 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 inadvertently passed a NULL pointer to a Windows API function and the function tried to access memory at the reference, a UAE or "general protection fault" would be generated.

Windows 95 provides parameter validations for all Win16–based APIs and checks incoming data to API functions to ensure that 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 that the data is within the proper range. If invalid data is found, an appropriate error number is returned to the application, and it is then up to the application to catch the error condition and handle it accordingly.

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

Robustness for Win32–Based Applications

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

Robustness support for Win32–based applications includes the following:

- A private address space for each running Win32–based application, segregating and protecting one application from others that are running concurrently
- Win32 APIs that support parameter validation and provide a stable and reliable environment
- Resource tracking by thread and the immediate freeing of resources when the thread ends

- Separate message queues for each running Win32–based application, ensuring that a hung Win32–based application does not suspend the entire system

A Private Address Space for Each Win32–Based Application

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

Parameter Validation for Win32 APIs

As with Win16–based applications, Windows 95 provides parameter validation for Win32 APIs used by Win32–based applications. The Windows 95 SDK 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 chapter.

Per-Thread Resource Tracking

Windows 95 tracks the resources allocated to Win32–based applications by thread. Unlike resources allocated to Win16–based applications, resources allocated to Win32–based applications are automatically released when a thread ends processing. This immediate freeing of system resources ensures that the resources are available for use by other running tasks.

Resources are cleaned up properly when threads either end execution on their own—for example, if the application developer inadvertently failed to free allocated resources—or when the user requests a local reboot that ends a given Win32–based application thread or process. Unlike Win16–based applications designed to run under Windows 3.1, Win32–based applications free up their allocated resources immediately when a separate thread or the application itself 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 to inform 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 addressed 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 would become full and accepted no new messages. Other Win16–based applications were then suspended until control was relinquished to them and they were able to check for event messages.

Windows 95 solves the problems inherent with the single message queue in Windows 3.1 by providing separate message queues for each running Win32–based application (each thread in a Win32–based application may have its own message queue.) As shown in Figure 38, the system takes messages from the input message queue and passes them 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 its incoming messages, other running Win16 and Win32–based applications are not affected.

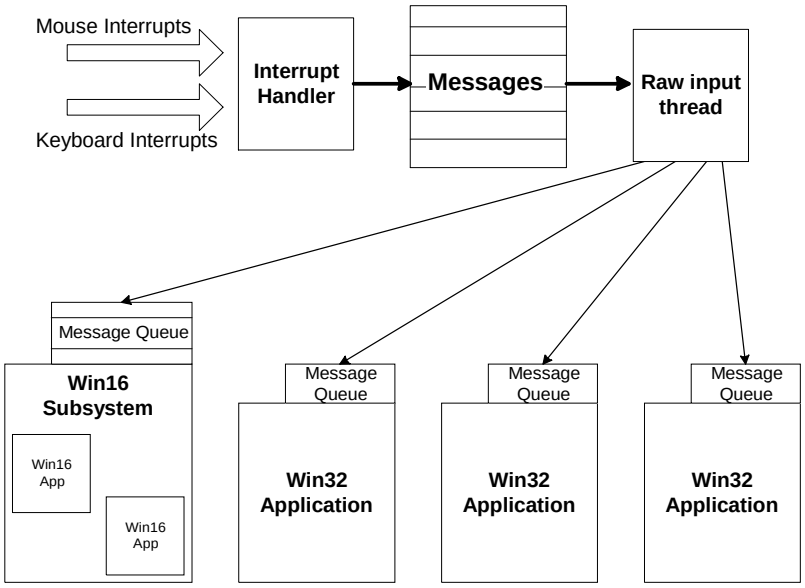

Figure 38. Win32–based applications use separate message queues for increased robustness

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

Local Reboot Effectiveness

Because of robustness improvements for Win32–based applications, including the use of a private address spaces, separate message queues, and resource tracking by thread, users should be able request a local reboot to end almost all ill-behaved Win32–based applications without affecting the integrity of the Windows system or other running applications.

When Windows 95 ends a Win32–based application, its resources are immediately deallocated and cleaned up by the system. Because Win32–based applications run in their individually allocated environments, this method is even more robust than the method for

reallocation of Win16–based application resources. For more details of the robustness of Win16-based applications, see the appropriate section earlier in this chapter.

Structured Exception Handling

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

The Win32 API supports a mechanism called *structured exception handling* for handling hardware-generated 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. By using this functionality, applications can identify and rectify error conditions that might occur outside their realm of control, providing a more robust computing environment.