ONE

Introduction

The advanced system services in the 32-bit API are the heart of Windows 2000 and Windows 98. You may have heard or read about many of them while learning about and experimenting with Windows 2000 and 98: function calls that execute over the network, multi-threading, standardized and heterogeneous network communications, the advanced C2 security system of Windows 2000, and so on. These are the cutting-edge capabilities that make both Windows 2000 and Windows 98 unique and interesting. The goal of this book is to show you how to use all of the Windows system services in your own applications.

The purpose of this chapter is to act as a road map and introduction. It also discusses several concepts that are of general interest because they are used constantly throughout the book. You may want to briefly scan this chapter now to see what is available, and then come back to specific sections as you need them.

1.1 One Hundred and Twenty-One Questions About Windows 2000 and 98

Many programmers coming to Windows 2000 and Windows 98 have questions about it. This section contains 121 of the most common system-programming questions, and information about where to look to find the answers. Use this list as a map for the rest of the book. These questions start with the most common and move toward the exotic.

1.1.1 General

1. How do I compile code on a Windows system? See Section 1.2 and Appendix A.
2. What are objects and handles? Why do they exist? See Section 1.5.
3. Why do there seem to be so many bugs in the 32-bit API? See Section 1.7.

1.1.2 File Operations

4. How do I read from and write to a file? See Sections 2.3 and 2.7.
5. How do I read and write a large quantity of data in a way that does not stall out my user interface? See Section 7.5 (overlapped I/O) or Chapter 6 (threads).
6. How do I move, copy, and delete files from within a program? See Section 2.5.
7. How do I call for the movement of a file, but postpone the actual move operation until the next time the system reboots because the file is currently in use? See Section 2.5.
8. How do I create a temporary file? See Section 2.6.
9. How do I create a temporary file that automatically deletes itself when the program is finished using it? See Section 2.7.
10. How do I map a file into memory to drastically improve read/write performance? See Section 2.11.
11. How do I read compressed files without decompressing them? See Section 2.10.
12. How do I uncompress a compressed file from within an application? See Section 2.10.
13. How do I tune system performance while reading certain files? For example, I have to read a large file from beginning to end, and it is flushing out the disk cache. How do I prevent that? See Section 2.7.
14. How do I access file information such as the size of the file, the last change date, and the attribute bits? See Section 2.4.
15. How do I lock whole files or individual records for exclusive or read-only access? See Section 2.9.

1.1.3 Drive Operations

16. How do I find out if a certain disk drive is mounted locally or over the network? See Section 3.3.
17. How do I find out which file system (FAT, HPFS, NTFS, etc.) a volume was formatted with? See Section 3.2.

18. How do I find out how much free space is available on a drive? See Section 3.4.

19. How do I determine which of the 26 drive letters are currently attached to a drive? See Section 3.5.

20. How do I change the volume label of a drive from within an application? See Section 3.6.

21. How can I create a list of all of the machines on the network? See Section 3.7.

22. How do I loop through all of the machines on the network and list all of the printers and drives they are currently sharing? See Section 3.7.

23. How do I form or cancel a connection to a remote drive or printer? See Section 3.7.

1.1.4 Directory Operations

24. How do I create and delete directories? See Section 4.2.

25. How do I set the current directory or find out the current directory? See Section 4.3.

26. How do I search the PATH and system directories for a file? See Section 4.4.

27. How do I search a directory for a file? See Section 4.5.

28. How do I recursively traverse all of the directories on a drive? See Section 4.5.

29. How do I detect changes to files and directories? For example, if a new file is added to a directory, is there an easy way to detect that immediately? See Section 4.7.

1.1.5 Active Directory—Answers Are in Section 5.4

30. How does a workstation discover its site?

31. How does a workstation find a directory server?

32. How do I log on?

33. What happens to access control lists on domain resources after migration?

34. What happens to ACLs when I delete a domain?

35. What happens to local groups?

36. When is the global catalog searched?

37. Do I have to use Microsoft's DNS service?

38. What are the advantages of using Microsoft's DNS service?

39. What happens to DHCP?

40. What happens to WINS?

1.1.6 Processes and Threads

41. How do I multi-thread an application? What does that mean? What is a thread? See Chapter 6.

42. How do I design an application so that it runs lengthy operations in the background rather than making the user wait? For example, my application has to recalculate an aerodynamic model and I do not want the user interface to hang for half an hour during the computation. See Chapter 6.

43. How can I take advantage of multiple threads to simplify the design of simulations that contain many separate and interacting entities? See Chapter 6.

44. How do I create separate threads that can handle high-priority events successfully without monopolizing the CPU? See Section 6.7.

45. How does the scheduling and priority system work? See Section 6.7.

46. How do I multi-thread an application so that it takes maximum advantage of multiple-CPU architectures in Windows 2000? See Section 6.5.

47. How do I launch a new separate and independent process from within an application? See Section 6.9.

48. How can I let two separate processes share data? For example, I want to be able to run an application that controls a service, so the application needs to be able to talk to the service. See Section 6.11.

49. How can I create a new process that inherits handles from its parent? See Section 6.10.

50. How do I assign local storage to each thread in a multi-threaded application? See Section 6.6.

51. How do I add multiple threads to an MFC program? See Section 6.4.

1.1.7 Synchronization

52. Why does Windows contain so many different synchronization mechanisms? Semaphores, Critical section, Mutexes—what good are they? See Chapter 7.

53. How do I guarantee that two threads do not access the same resource at the same time? See Chapter 7.

54. I have a multi-threaded program that occasionally locks up for no apparent reason. What is causing this problem? See Section 7.3.

55. How do I integrate synchronization mechanisms into an MFC program without shutting down the event loop? See Section 7.7.

1.1.8 Network Communications

56. How can I create an application that can talk with other copies of the same application, or other different applications, on the network? See Chapter 8.

57. How do I broadcast information on the network so that all of the machines on the network see it? See Section 8.3.

58. How do I create point-to-point network connections between two Windows machines? See Section 8.4.

59. How do I create client/server architectures? What code and techniques are necessary to create the server and the client? See Section 8.5.

60. How do I communicate with UNIX and other TCP/IP machines using UDP and TCP packets? See Section 8.6.

61. How do I read and write to the same pipe in a multi-threaded application? See Section 8.5.

1.1.9 Remote Procedure Calls

62. How do I use Remote Procedure Calls? What is a Remote Procedure Call? See Chapter 9.

63. How do I design a program so that it effectively uses Remote Procedure Calls? How do I know when and when not to use RPCs? See Section 9.3.

64. How do I create client/server architectures using RPCs? See Chapter 9.

65. How do I create an IDL file and use the MIDL compiler? See Chapter 9.

66. How do I create an RPC server in Windows? See Section 9.5.

67. What is the RPC name server? How do I register an RPC server that I create with the name server? See Sections 9.6 and 9.8.

68. How do I create an RPC server that runs on its host machine all of the time, starting when the host boots? See Section 12.11 and Chapter 12.

69. How do I call an RPC from an RPC client program? See Chapter 9.

70. Can an RPC client on a Windows machine access an RPC server on a UNIX machine? What about the reverse? See Section 9.8.

71. What is the difference between automatic and manual binding in RPCs? See Chapter 9.

1.1.10 Serial Communications

72. How do I use communications ports in Windows? I want to create a BBS or a terminal emulator. See Chapter 14.

73. How do I send and receive characters from a communications port? See Section 11.4.

74. How do I set communications parameters like baud rate and parity on a communications port? See Section 11.4.

75. How do I manage different flow control policies on a communications port? See Section 11.6.

76. How do I easily communicate with other Telnet servers on the network? See Section 11.7.

1.1.11 Windows 2000 Services

77. How do I run an application in the background under Windows 2000, starting when the machine boots and regardless of who is logged on? For example, I need to run a background data-logging program in my lab, but I don't want to tie up the entire machine. See Chapter 12.

78. How do I integrate a service into the Services applet of the Windows 2000 Control Panel? See Section 12.5.

79. How do I create multiple service threads that share global variables? See Section 12.7.

80. How can a service communicate with the user? See Section 12.5.

1.1.12 Security

81. How does the Windows 2000 security system work? See Chapter 13.

82. How do I change the security attributes on a file from within one of my applications? See Sections 13.4 and 13.8.

83. How do I secure a named pipe so that only certain users on the network can access it? See Sections 13.4 and 13.9.

84. How do I un-secure a named pipe so that any user can access it? See Section 13.9.

85. How do I create a named pipe server than any user on the network can access, but preserve security at the same time? Is there a way for a named pipe server to impersonate a connected user? See Section 13.9.

86. How do I access the auditing features of the security system so that accesses to certain objects get logged in the event viewer? See Section 13.2.

87. What are ACL, DACL, SACL, SID, and ACE? See Section 13.3.

88. How do I modify existing security descriptors? See Section 13.8.

89. How do I set the system time or shut down the system? Even though I'm the Windows 2000 administrator, the system functions for these operations fail when I run the code. See Section 13.7.

90. What is a privilege? How do I enable and disable privileges? See Section 13.7.

1.1.13 Consoles

91. How do I port a text program (for example, a program based on the curses library in UNIX or a vt-100 terminal emulator) over to Windows? See Chapter 14.

92. How do I access keystrokes from the keyboard one character at a time (raw input) in a console program? See Section 14.4.

93. How do I add streaming text output to a normal Windows or MFC program to help with debugging? See Section 14.3.

1.1.14 System Functions

94. How do I find out the number of buttons on the mouse? See Section 16.2.

95. How do I find out the screen width and height? See Section 16.2.

96. How do I get and set system colors? See Section 16.2.

97. How do I find out the number of processors in the machine? See Section 16.1.

98. How do I find out the machine's name on the network? See Section 16.2.

99. How do I find out what version of Windows is currently running? See Section 16.2.

100. How do I get and set environment variable information? See Section 16.3.

101. How do I shut down the system or log off the current user? How do I shut down remote machines on the network? See Section 16.4.

1.1.15 Dynamic Link Libraries

102. What is a Dynamic Link Library? What are the advantages of DLLs? See Chapter 17.

103. How do I create, compile, and load a DLL? See Sections 17.3 and 17.4.

104. What is the difference between run-time and load-time linking? What are the advantages of each approach? See Section 17.5.

105. How do I distribute a DLL so that other programmers can use it? See Chapter 17.

106. What is a DLL entry point? How can I use one to improve memory management? See Section 17.6.

107. How do I create global and dynamic variables that are shared or private inside a DLL? See Section 17.7.

1.1.16 Miscellaneous

108. What is the registry? See Section 21.1.

109. Where is the best place to store configuration information for my application? See Section 21.1.

110. How do I add a new key to the registry? See Section 21.1.

111. How do I read and write registry values? See Section 21.1.

112. How do I store events from my own applications in the event log? See Section 21.2.

113. How do I determine which events belong in the event log? See Section 21.2.

114. How do I create strings for my event log messages? See Section 21.2.

115. How do I get and set the current system time? See Section 21.3.

116. How do I find out what time zone the current machine is in? See Section 21.3.

117. How do I allocate and free memory at the system level? See Section 21.4.

118. How can I play sounds other than a simple beep during errors? See Section 21.6.

119. When creating my own libraries, how can I make their functions conform to Windows' GetLastError format? See Section 21.6.

120. What is structured exception handling? How can I use it in my programs? See Section 21.5.

121. How do I contact the author? See Appendix B.

1.2 Compiling Code

This book contains hundreds of listings. You will find that these listings break down into four different categories:

1. Pure console programs that read and write text data
2. GUI programs using the 32-bit API

3. GUI programs using MFC
4. Other programs: DLL code, RPC code, and so on

Appendix A contains information on compiling the first three types of code. The fourth type is covered when the code appears. For example, Chapter 9 on RPCs contains makefiles for using the MIDL compiler and for compiling RPC programs.

1.3 Terminology

Windows, like UNIX or VMS, is a complete operating system. When you use this operating system, you refer to it as "Windows 2000," or "Windows 98," or generically as "Windows," and leave it at that. When you program for the system, you are programming for Windows as well. More specifically, however, you are using the "Win32 API" (Application Programmer Interface), also known as "the 32-bit API." In this book, programming for "Windows," "Win32," and "the 32-bit API" are equivalent, and you will find them used interchangeably.

The 32-bit API is huge. It contains thousands of functions broken up into hundreds of categories. It is a comprehensive, all-encompassing programmer interface to the many capabilities provided by the Windows operating system. The API is the same whether you are using Windows 2000 or Windows 98. However, a number of functions will report "not implemented" errors in Windows 98 because they are not supported. See Section 1.8 for a description of the differences between Windows 98 and Windows 2000.

The 32-bit API is so called to distinguish it from the older 16-bit API used by standard Windows 3.1 running on top of MS-DOS. The 16-bit API contains all of the Windows and Graphics functions found in the 32-bit API (that is why most older Windows programs port to Windows 2000 and 95 so easily), but the 16-bit API does not have any of the system services found in the 32-bit API. The 32-bit API also includes a number of operations normally found in the standard C libraries, in an attempt to centralize things. For example, the 32-bit API includes functions to move, zero, and copy blocks of memory (see Section 21.4).

Win32s is a subset of the 32-bit API. It runs on DOS machines, and provides a way to execute Win32 applications in the older Windows environment. Win32s is missing about half of the system services. For example, it does not support threads, services, event logging, or RPCs, etc. You can call these functions in Win32s, but they will return error codes.

This book deals with the system services of the 32-bit API. The system services are the core of the operating system. They give you access to the modern capabilities that make the 32-bit versions of Windows interesting:

- Processes and threads
- Synchronization
- Remote Procedure Calls (RPCs)
- Event logging
- Network communications
- TCP/IP networking with UNIX and other machines
- File mapping
- Interprocess communication
- Compressed file access
- Network drive and printer sharing
- Serial communications
- Services (background processing like UNIX daemons)
- Object and file security

The other two major sections of the API are the Graphics Device Interface (GDI) for drawing and printing, and the Window Management functions for creating GUIs. Both of these sections are encapsulated by MFC, which is described below. Because of MFC, you generally do not access either of these sections directly when creating Windows applications in C++.

You will generally write and compile your Win32 programs using either "the Microsoft Win32 SDK" or Visual C++. The SDK is a command-line environment, while Visual C++ is a graphical environment.

Visual C++ (but not the SDK) ships with a C++ class library called the Microsoft Foundation Class library, or MFC. MFC makes the creation of GUI programs much easier. It encapsulates all of the window-management and GDI functions available in the 32-bit API. Another book in this series is called "Visual C++: Developing Professional Applications for Windows 98 and Windows 2000 using MFC," and it covers MFC programming in detail. Many of the system-services examples in this book show you how to apply the concepts to MFC code. For example, Chapter 6 contains examples that show you how to multi-thread MFC programs. Chapter 7 shows you how to handle semaphores, mutexes, and events in MFC code, and Chapter 9 applies RPCs to an MFC program.

1.4 Error Handling

Most functions in the 32-bit API return error status in two different ways. Many functions return the first indication of an error in the return value for the function. For example, the **RemoveDirectory** function in Chapter 4 is typical:

RemoveDirectory	*Removes an empty directory*
`BOOL RemoveDirectory(` `  LPTSTR dirName)`	
DirName	Name/path of the directory to remove
Returns TRUE on success	

The **RemoveDirectory** function returns a boolean value that is TRUE if the function was successful and FALSE otherwise. Other functions return a specific value to indicate an error:

OpenService	*Opens a connection to the specified service*

`SC_HANDLE OpenService(`
`  SC_HANDLE scm,`
`  LPCTSTR name,`
`  DWORD access)`	

scm	A handle to the SCM from **OpenSCManager**
name	The name of the service used internally
access	The desired access
Returns a handle to the service or NULL on error	

The **OpenService** function returns 0 when something goes wrong. A typical use of the **GetLastError** function is shown below:

```
success = RemoveDirectory(s);
if (!success)
   cout << "Error code = " << GetLastError()
      << endl;
```

GetLastError	*Returns the error code of the most recent error*
`DWORD GetLastError(VOID)`	
Returns the error code of the last error	

When the return value indicates an error, you can obtain a numeric error code identifying the exact problem by calling the **GetLastError** function. **GetLastError** returns an integer that you can look up in the "Error Codes" section of the API help file (see Section 1.6). For example, in the example above the error value might be five. When you look up that value in the Error Codes section, you will find that it translates to "Access Denied," which means that the file system's security features will not allow you to delete the specified directory.

1.5 Handles and Objects

Windows uses the concepts of a handle and an *object* throughout the system services. An object is owned by the operating system and represents a system resource. For example, a thread is represented as an object. The operating system stores all information about the thread in the thread's object. The object is opaque to the programmer.

A handle points to an object. It is possible to create a single object and refer to it with several different handles. You can think of the handles as pointers that let you access the object.

The 32-bit API defines the following "kernel objects." All of them return a handle when they are created or opened:

Object	Function	Chapter
Access tokens	OpenProcessToken	Chapter 13
	CloseHandle	
Console	CreateConsoleScreenBuffer	Chapter 14
	CloseHandle	
Console device	GetStdHandle	Chapter 14
	CloseHandle	
Communication port	CreateFile	Chapter 11
	CloseHandle	
Event	CreateEvent	Chapter 7
	CloseHandle	
Event log	OpenEventLog	Chapter 21
	CloseEventLog	
File	CreateFile	Chapter 2
	CloseHandle	
File change	FindFirstChangeNotification	Chapter 7
	FindCloseChangeNotification	
File mapping	CreateFile	Chapter 2
	CloseHandle	
Find file	FindFirstFile	Chapter 4
	FindClose	
Mailslot	CreateMailslot	Chapter 8
	CloseHandle	
Mutex	CreateMutex	Chapter 7
	CloseHandle	

Object	Function	Chapter
Named pipe	CreateNamedPipe	Chapter 8
	CloseHandle	
Pipe	CreatePipe	
	CloseHandle	
Process	CreateProcess	Chapter 6
	CloseHandle	
Semaphore	CreateSemaphore	Chapter 7
	CloseHandle	
Thread	CreateThread	Chapter 6
	CloseHandle	

Of these objects, some are global. For example, there is one instance of the event log and any process can open a handle to it. Some are private. For example, a Find File handle is private to the process that opens it. Others can be shared among processes:

- Process
- Thread
- File
- File mapping
- Event
- Semaphore
- Mutex
- Named pipe
- Mailslot
- Communication device

For example, one thread or process might create a semaphore, and other processes and threads can open handles to it so that they can all use it for synchronization. The single semaphore object is shared by the all of the processes and threads. It is also possible to duplicate and inherit (Chapter 7) handles to these objects.

Most objects that return a handle can be secured using the Windows 2000 security system. See Chapter 13 for a complete list and details. Security is a primary reason for this "object" model. The other reason for it is that it adds a level of abstraction to the operating system, which keeps programmers from knowing too much. A HANDLE is a generic 32-bit pointer. It tells you, as a programmer, nothing about the object it points to, so you are unable to get into the low-level structures and code directly to them. This allows the operating system to change over time without its designers having to worry about code that breaks the rules.

If you want to open a file and access it through the 32-bit API, you use the **CreateFile** function (see Chapter 2) as shown below:

Listing 1.1 *An API-level program that reads from a file and writes to it*

```
// file3.cpp

#include <windows.h>
#include <iostream.h>

void main()
{
   HANDLE fileHandle;
   BOOL success;
   char s[10];
   DWORD numRead;
   char filename[MAX_PATH];

   // get the file name
   cout << "Enter filename: ";
   cin >> filename;

   // Open the file
   fileHandle = CreateFile(filename, GENERIC_READ,
      0, 0, OPEN_EXISTING, 0, 0);
   if (fileHandle == INVALID_HANDLE_VALUE)
   {
      cout << "Error number " << GetLastError()
         << " occured on file open." << endl;
      return;
   }

   // Read from file until eof, writing to stdout
   do
   {
      success = ReadFile(fileHandle, s, 1,
         &numRead, 0);
      s[numRead] = 0;
      cout << s;
   }
   while (numRead>0 && success);

   // Close the file
   CloseHandle(fileHandle);
}
```

Note that the call to **CreateFile** returns a HANDLE to the variable **fileHandle**. This variable points to a *file object* that talks to the actual file on the

disk. The file object in this case remembers the state of the file. For example, it remembers where the file pointer is in the file. The code in Listing 1.1 uses the **fileHandle** in the call to **ReadFile** to read from the file, and then again in the call to **CloseHandle** to close the file.

As another example, let's say that you want to use a mutex object (see Chapter 7) to control access to a global resource. The program calls **CreateMutex** to create a mutex object, and the function returns a handle to the object. The object is owned by the operating system and remembers the state of the mutex, while the handle lets your program reference the object to query and change the state using functions like **WaitForSingleObject** and **ReleaseMutex** (see Chapter 7). When you create the mutex you can give it a name.

Now another program can call **OpenMutex** with that same name. The process will receive a handle that refers to the same operating system object. It can query the mutex and change its state, so now the two programs can synchronize their behavior using the single mutex object managed by the operating system. Once *both* processes close their handles to the object, the operating system deletes the object.

For more information on objects and handles, see Chapters 7 and 13, and the "Handles and Objects" section in the documentation. See the following section for a discussion on the documentation.

1.6 Using the Microsoft Documentation

The 32-bit API comes with quite a bit of documentation. It is available both on-line and on paper. On paper it is referred to as the Win32 Programmer's Reference books (five volumes). On-line it is in a help file named `api32.hlp`. The on-line version is generally much easier to use, because of both the hypertext links and the Search button (especially the NEAR capability), which lets you find any word in the entire file instantly.

The goal of this book is not to duplicate the information already available in these references. That would be impossible. Here, the goal is to help you see how to use the functions available in the API to solve problems. Therefore, this book is full of concise, easy-to-understand example code and explanations that help you understand how to use the capabilities of the 32-bit API in your own programs.

You should use Microsoft's 32-bit API documentation as a supplement to this book. When you are working through an example, look up each function or structure in the API documentation so that you can see what all the parameters and members actually do. See Section 1.7 for an explanation of why this can be important. The explanations in the on-line help file are generally quite clear, and will make sense in the context of the example. Click over to some of the related structures and functions to read about them as well. You can rapidly master the API this way.

1.7 Bugs in the 32-Bit API

Occasionally you will write a new Windows program, and it simply will not work the way you expect it to work. You will write the program and run it, and it will crash. You will look at the code and diddle with it a little bit, and the problem will remain. After 15 minutes, your response to this improbability is going to be, "Wow, I found a bug in the 32-bit API. I can't believe that they don't test this stuff more carefully." At dinner that night you will rail at your spouse about the inept programmers at Microsoft.

That is the natural human reaction. However, I'd say that in 99.9% of cases, the problem lies in your code rather than in the API. In many cases, the problem lies in your perception of how you *think* the code should work, even though the documentation clearly states otherwise and you skipped over or missed that part.

To let you see just how easy it is to create problems when working with the API, let me show you five of my own bugs. These bugs all arose while developing the sample code for this book. You, of course, won't make mistakes this stupid, but they demonstrate some of the possibilities. In each one of the five cases, it seemed at first that there was a mysterious and unknown bug somewhere deep in the bowels of the API, but it later turned out to be a stupid mistake on my part.

1. In Listing 6.2, the original code did not contain the keyword **volatile**. The program worked fine as long as there was a call to **Sleep** or some other API function inside the thread function, but as soon as I removed the call to **Sleep**, **count** remained resolutely at zero. However, if I turned on debugging, the code worked just fine. Obviously there was a bug in the thread-handling portion of the API causing **count**'s faulty behavior. However, it turned out to be compiler optimizations that caused the **count** variable to get placed in a register. Adding the **volatile** keyword turns off those optimizations, so the code works fine.
2. When developing the inheritance example in Listings 6.13 and 6.14, it seemed to be impossible to get the child process to inherit anything. I would run the code, and the child would display an "invalid handle" error every time. In some cases, the behavior was even more bizarre: The child process would get created over and over again, even though there are no loops anywhere in the code. Obviously there was a bug in the handle-inheriting portion of the API. However, it turned out that I was passing the **sa** parameter to **CreateProcess** as the last parameter rather than as the fourth. Changing the erroneous parameter list eliminated the problem. The **sa** parameter *is* last in the call to **CreateNamedPipe**, but not to **CreateProcess**.

3. In the multi-threaded client program shown in Listing 8.11, the code simply refused to handle the incoming pipe data correctly. Either the incoming data was ignored completely, or the program would read one piece of data and then stop, or, if the overlapped structures were made global, the program would work with one client but then mysteriously and continuously loop in the server whenever a second client connected. Obviously it was a serious bug in the way overlapped I/O works with named pipes. However, it turned out that I was not creating the events and placing them in the overlapped structures to properly initialize them. I was thinking that because I didn't use the events anywhere in the code, I did not need to create them. Once I created the events correctly, the code worked just fine.
4. In the bounded buffer example using semaphores in Listing 7.8, the first version I created just would not work at all. It would terminate immediately, or run a little and die, without any hint of an error message. Nothing seemed to be wrong—it just would up and die for no apparent reason. Obviously the semaphore portion of the API was seriously flawed, or the mixing of two different synchronization mechanisms led to internal interactions that rendered the entire API inoperable. However, it turned out that I was placing the two thread handles into `handles[1]` and `handles[2]` rather than into `handles[0]` and `handles[1]`. Fixing that minor problem had a wonderful effect on the code.
5. I worked on the simple **push** and **pop** functions in Chapter 7 very early in my Windows career. I happen to like synchronization problems and I wanted to try out critical sections. The test code would run a tiny bit and then hang. I'd run it again and it would hang again. Every now and then it worked fine, but then it would hang again in the next test. Nothing seemed to fix the problem. I tried everything, but adding in the critical sections just destroyed the ability of the program to execute. Obviously the critical section portion of the API was totally flawed. Apparently Microsoft had added it at the last minute and never bothered to test it. However, it turned out that I had forgotten to place a call to **LeaveCriticalSection** before the first return in the **pop** function, so the code could not proceed if that branch got executed. Adding the function call solved the problem.

As you can see, I make my fair share of mistakes, just like everybody else. For the record, let me state that I have written and helped test thousands and thousands and thousands of lines of Windows code, and I have seen many hundreds of "mysterious API bugs." However, I have *never* seen one that did not go away once the programmer wrote the code correctly. It's something to keep in mind when one of your programs is misbehaving. Read

the manual, and look carefully at your code. The bug is in there somewhere, staring back at you.

1.8 Differences Between Windows 98 and Windows 2000

Windows 2000 and Windows 98 are very similar at the API level. In fact, a good way to think about Windows 98 is to think of it as "Windows 2000 Lite." Windows 98 does most of the things that Windows 2000 can do, but there are several capabilities that Windows 2000 has that Windows 98 lacks. In particular, Windows 98 is missing parts of Windows 2000 security system (Chapter 13), Windows 2000 services (Chapter 12), and some Windows 2000 network functionality (Chapter 8). A good way to understand these differences is to compare Windows 2000, Windows 98, and Windows 3.1 and see how they evolved from one another.

Windows 3.1 and Windows for Workgroups 3.1 were built on top of MS-DOS, the Microsoft operating system for PC-class machines. This layering is a liability because MS-DOS is primitive by today's standards. MS-DOS has a very simple file system with limited features and no security. MS-DOS offers none of the capabilities the user would expect to find in a "real" operating system—features such as virtual memory, multiple processes, interprocess communication, and so on. Windows 3.1 takes care of some of these problems itself as best it can. For example, it offers a good memory management system and cooperative multi-tasking. But because it is built on DOS, the system is fragile, and the file management facilities are poor. The system is also permanently attached to PC-compatible hardware. There is really no easy way to move it to other platforms.

Windows 2000 is Microsoft's answer to these problems. It is a complete and modern operating system built from the ground up as a total solution to workstation computing. Windows 2000 offers everything you would expect to find in a modern operating system:

- 32-bit instructions and memory addressing: Like any other modern workstation operating system, Windows 2000 uses a 32-bit numeric and address format. Unlike older DOS and Windows machines, memory addressing is "flat," so you can create arrays as large as you like in memory.
- Preemptive multi-tasking (processes and threads): Microsoft Windows 3.1 uses *cooperative multi-tasking*, in which applications yield the processor to one another at each application's discretion. Windows 2000 instead uses *preemptive multi-tasking*. Under this system the OS automatically allocates CPU time to each application (called a *process*). Because the OS, rather than the individual applications, is in control of CPU time slices, the system is extremely stable and

robust. In addition, Windows 2000 applications can divide themselves up into separate *threads* of execution. An application might do this in order to perform a lengthy calculation in the background without affecting the user interface. The OS can schedule threads independently.

- Symmetric Multi-processing: A Windows 2000 machine can contain more than one CPU. In a multi-processor machine, Windows 2000 will allocate different threads to different CPUs to take full advantage of all CPU power available. In addition, Windows 2000 is itself multi-threaded, and its different threads can run on separate processors.
- Multiple platforms (Intel, MIPS, Alpha, etc.): Windows 2000 is designed to run on a variety of processor architectures. Unlike UNIX, which looks different depending on whose hardware it is running on, Windows 2000 looks exactly the same to both the user and the programmer, no matter which sort of architecture is running it.
- Remote procedure calls: Windows 2000 supports OSF-style Remote Procedure Calls (RPCs), which can be used to easily build client/server applications that run efficiently on a network.
- Total network support (TCP/IP, NetBEUI, etc.): Windows 2000 is designed to run on a network, and supports a native format for communication with other Windows 2000 machines as well as machines running TCP/IP protocols.
- C2 certified security: When using the Windows 2000 File System (NTFS), Windows 2000 creates a secure system. All users must log in with an account name and password. Files and directories on the disk are protected by access control lists (ACLs). Applications are isolated from one another in memory so that the crash of one does not affect any of the others.

Microsoft has covered all of the power in Windows 2000 with the familiar Windows-type interfaces found in the other Windows products. This decision means that anyone familiar with previous versions of Windows can use Windows 2000 very quickly. From an administrative standpoint, Windows 2000 contains many new features. Windows 2000 attempts to completely separate use of the system from administration of the system so that users do not have to waste their time on administrative tasks. A user may "own" the machine on his or her desktop, but that does not mean that the user wants to spend time backing up the hard disks, installing software, or configuring the network. Those administrative tasks are better left to a trained administrator. Windows 2000 therefore provides separate administrative tools that allow remote access. For example, if a machine is sitting on a network in a large company, an administrator can attach to the machine over the network to update software, change accounts, and so on. Windows 98 contains very little in the way of administrative features.

From a programmer's standpoint, Windows 2000 uses the Win32 API, which contains a wealth of advanced operating system services. These services are the subject of this book.

Windows 98 contains most of the API features of Windows 2000. The Win32 API in Windows 2000 and Windows 98 are identical, and Windows 98 therefore uses the same process and thread model, the same file-access functions, the same network-access functions, and so on. Most of the differences between Windows 2000 and Windows 98 are omissions. Certain features were removed from Windows 98 to improve its performance on low-end hardware. The list below summarizes these differences:

- Symmetric Multi-processing: Windows 98 works only on single-CPU hardware architectures.
- Multiple platforms: Windows 98 currently runs only on Intel hardware.
- C2 certified security: Windows 98 contains only part of Windows 2000's advanced security features.

You will learn about other, more subtle differences in the process of reading this book.

In general, you can think of Windows 2000 as a complete workstation operating system intended for use in corporate, university, and engineering environments. Windows 98 is a lower-end product intended more for home and small-business applications.