UNIX Custom Application Migration Guide

Volume 3: Migrate Using Win32/Win64

Published: May 2006

Contents

About This Volume

Introduction to Volume 3

Volume 1 of the *UNIX Custom Application Migration Guide* discussed how to apply the Envisioning and Planning Phases of the Microsoft® Solutions Framework (MSF) Process Model when conducting a UNIX to Microsoft Windows® migration project. This volume, Volume 3: *Migrate Using Win32/Win64*, applies the next phases in the Process Model, the Developing Phase and the Stabilizing Phase, and directs it specifically for using the Microsoft Win32® and Win64 APIs. This volume describes the architectural and potential coding differences between the UNIX and Windows environments and discusses various ways to implement these differences in the Windows environment using the Win32/Win64 API. This volume addresses these potential coding differences by looking at the solution from various categories.

These categories are:

- Process management.
- Thread management.
- Memory management.
- File management.
- Infrastructure services.
- User interface migration.
- Fortran code migration.
- Deployment considerations and testing activities.
- Stabilizing Phase activities.

For each of these categories, this volume:

- Describes the coding differences between UNIX and Windows.
- Outlines options for converting the code using Win32/Win64 API.
- Illustrates the options with source code examples.

This information will help you choose the solution that is appropriate to your application. Sufficient code examples and references are provided in this volume to aid you in the migration process. You can also refer to the Platform SDK documentation to obtain more details on the Win32/Win64 API.

Additional information on activities in the Developing Phase as they relate to a migration project is available in Chapter 2, “Developing Phase: Process Milestones and Technology Considerations” and Chapter 8, “Developing Phase: Deployment Considerations and Testing Activities” of this volume.

Intended Audience

The technical information in this volume is provided to support the activities undertaken during the Developing Phase of a migration project. It is intended for developers and testers who are involved in migrating UNIX code to Windows using the Windows API. Using the guidance provided in this volume, a UNIX programmer will learn how to modify code so that it can be recompiled to run in a Windows environment using the Windows API, and a Windows programmer will learn how to port UNIX functions to Windows.

The specific advantages that this volume provides developers and testers are as follows:

- **Developers**. Developers can learn about the various alternative methods for migrating from UNIX to Windows using the Win32/Win64 API and how to choose the best strategy to fit their environment and the application types.
- **Testers**. Testers can gain more insight on the testing methodology that is best suited for their migration scenario. With the help of this guide, they can test the application for various aspects, such as functionality, management, performance, and stability.

Knowledge Prerequisites

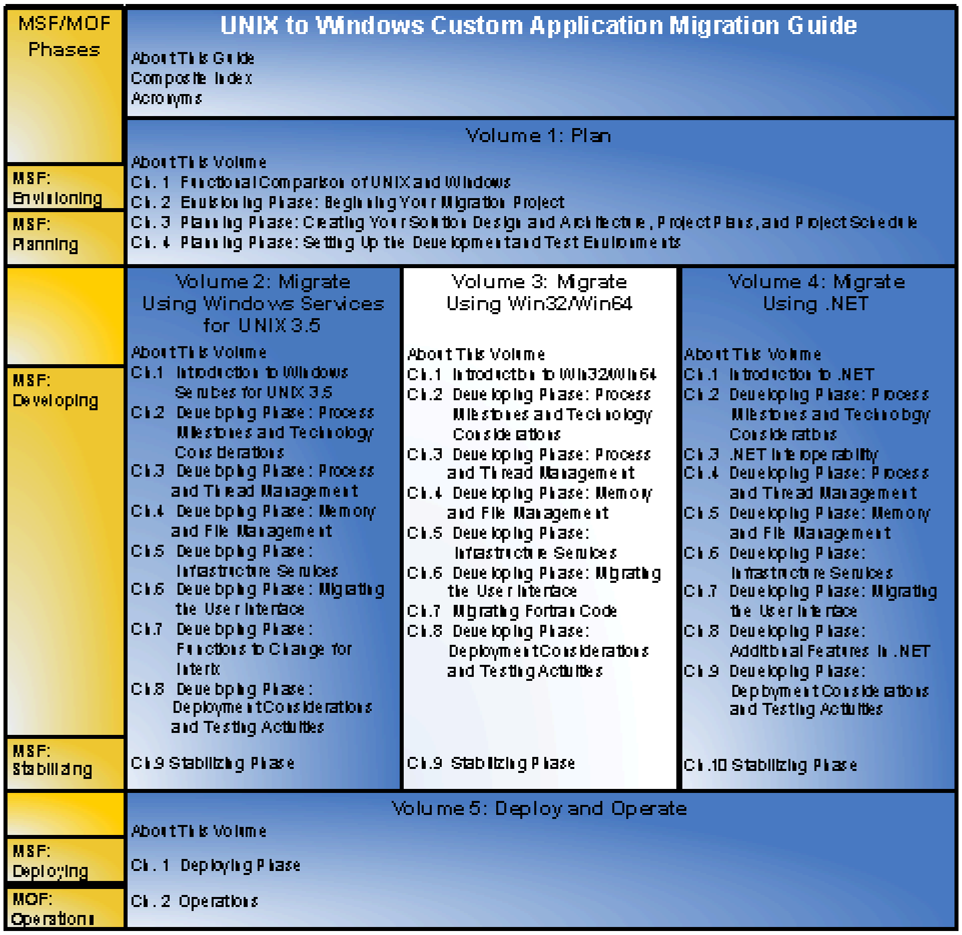
The readers of this volume should possess the following knowledge prerequisites:

- Basic knowledge of UNIX and Windows internals such as process and thread management, file and memory management, and various infrastructure services features.
- Hands-on experience on Windows environments.
- Familiarity with UNIX administration skills.
- Familiarity with development involving Win32/Win64 API in a Windows environment.

It is also recommended that you read the “About This Guide” section in Volume 1: *Plan* as well as the rest of the *Plan* volume before reading this volume.

Layout of the Guide: Volume 3

The following diagram depicts the layout of the guide and how the volumes of the guide correlate with the components of the MSF Process Model. The white-shaded portion indicates the position of the current volume in the layout of the entire guide.



MSF/MOF Phases

UNIX to Windows Custom Application Migration Guide
About This Guide
Composite Index
Acronyms

MSF: Envisioning
MSF: Planning

Volume 1: Plan
About This Volume
Ch. 1 Functional Comparison of UNIX and Windows
Ch. 2 Envisioning Phase: Beginning Your Migration Project
Ch. 3 Planning Phase: Creating Your Solution Design and Architecture, Project Plans, and Project Schedule
Ch. 4 Planning Phase: Setting Up the Development and Test Environments

MSF: Developing
MSF: Stabilizing

Volume 2: Migrate Using Windows Services for UNIX 3.5
About This Volume
Ch.1 Introduction to Windows Services for UNIX 3.5
Ch.2 Developing Phase: Process Milestones and Technology Considerations
Ch.3 Developing Phase: Process and Thread Management
Ch.4 Developing Phase: Memory and File Management
Ch.5 Developing Phase: Infrastructure Services
Ch.6 Developing Phase: Migrating the User Interface
Ch.7 Developing Phase: Functions to Change for Interix
Ch.8 Developing Phase: Deployment Considerations and Testing Activities
Ch.9 Stabilizing Phase

Volume 3: Migrate Using Win32/Win64
About This Volume
Ch.1 Introduction to Win32/Win64
Ch.2 Developing Phase: Process Milestones and Technology Considerations
Ch.3 Developing Phase: Process and Thread Management
Ch.4 Developing Phase: Memory and File Management
Ch.5 Developing Phase: Infrastructure Services
Ch.6 Developing Phase: Migrating the User Interface
Ch.7 Migrating Fortran Code
Ch.8 Developing Phase: Deployment Considerations and Testing Activities
Ch.9 Stabilizing Phase

Volume 4: Migrate Using .NET
About This Volume
Ch.1 Introduction to .NET
Ch.2 Developing Phase: Process Milestones and Technology Considerations
Ch.3 .NET Interoperability
Ch.4 Developing Phase: Process and Thread Management
Ch.5 Developing Phase: Memory and File Management
Ch.6 Developing Phase: Infrastructure Services
Ch.7 Developing Phase: Migrating the User Interface
Ch.8 Developing Phase: Additional Features in .NET
Ch.9 Developing Phase: Deployment Considerations and Testing Activities
Ch.10 Stabilizing Phase

MSF: Deploying
MOF: Operations

Volume 5: Deploy and Operate
About This Volume
Ch. 1 Deploying Phase
Ch. 2 Operations

Figure 0.1. UCAMG organization

Organization of Content

The content of this volume is organized into the following chapters:

- **About This Volume**. This chapter provides information on the organization of the guide and about its intended audience. It also lists the knowledge prerequisites required for this volume and provides resources, such as document conventions, used in this guide.
- **Chapter 1: Introduction to Win32/Win64**. This chapter describes how you can modify the source code of your UNIX applications so that the code compiles on the Microsoft Windows operating system using the Microsoft Win32 and Win64 application programming interfaces (APIs). This chapter also describes the improvements that Win64 offers over Win32.
- **Chapter 2: Developing Phase: Process Milestones and Technology Considerations**. This chapter introduces the Developing Phase and discusses using Microsoft Visual Studio® .NET 2003 and the Platform Software Development Kit (SDK) for the Developing Phase. The chapter includes a section about how to make your code compliant with both 32-bit and 64-bit architectures.
- **Chapter 3: Developing Phase: Process and Thread Management**. This chapter discusses the similarities and differences in the implementation of process and thread management in UNIX and Microsoft Windows operating systems. The chapter first discusses the UNIX and Windows process management mechanism and the Windows APIs related to processes. It then discusses threads and their implementation in Windows.
- **Chapter 4: Developing Phase: Memory and File Management**. This chapter discusses the similarities and differences in memory and file management between the Microsoft Windows API and UNIX. It also provides various memory and file management-related functions and APIs that are available in both environments.
- **Chapter 5: Developing Phase: Infrastructure Services**. This chapter discusses the potential coding differences between UNIX and Microsoft Windows operating systems with respect to infrastructure services. The infrastructure services discussed in this chapter include security, handles, error and exception handling, signals versus events, interprocess communication, and networking.
- **Chapter 6: Developing Phase: Migrating the User Interface**. This chapter describes how to migrate from a UNIX-based user interface to a Microsoft Windows-based user interface. Because the majority of UNIX graphical interfaces are built on X Windows and Motif, the chapter focuses on porting code from X Windows to the Windows operating system.
- **Chapter 7: Developing Phase: Migrating Fortran Code**. This chapter examines the process of migrating a Fortran code to the Microsoft Windows operating system from the UNIX environment. This migration may be to either the Windows subsystem or the Interix subsystem.
- **Chapter 8: Developing Phase: Deployment Considerations and Testing Activities**. This chapter discusses the key aspects of deploying and testing a migrated application on Microsoft Windows operating systems. You will also be able to identify the deployment requirements, such as packaging and deploying of tools and administering the deployed Microsoft Win32 applications. This chapter also discusses various testing activities that you need to carry out in the Developing Phase.
- **Chapter 9: Stabilizing Phase**. This chapter discusses the different levels of testing and tuning that must be administered to the applications that are migrated to Windows using Windows Services for UNIX 3.5.

Resources

This section describes the various resources that are included in the *UNIX Custom Application Migration Guide* and information that will assist in using the guide.

Acronyms

See the Acronyms list accompanying this guide for a list of the acronyms and their meanings used in this volume.

Document Conventions

The document conventions used in this volume are primarily designed to help you quickly identify the operating system and the interface (command line or graphical) being discussed—Windows or UNIX. In general, Windows operating-system commands are executed by clicking user interface (UI) elements, and these elements are visually distinguishable by the use of bold text. In contrast, the UNIX operating system typically uses a command-line interface, and these instructions are visually distinguished by the use of a Monospace font.

These interface and execution differences are not absolute; and in case visual cues do not unambiguously distinguish between the two operating systems, the text will clearly make this distinction. Table 0.1 lists the document conventions used in this volume.

Table 0.1. Document Conventions

Text element	Meaning
Bold text	Used in the context of paragraphs for commands; literal arguments to commands (including paths when they form part of the command); switches; and programming elements, such as methods, functions, data types, and data structures. Also used to identify the UI elements.
Italic text	Used in the context of paragraphs for variables to be replaced by the user. Also used to emphasize important information.
Monospace font	Used for excerpts from configuration files, code examples, and terminal sessions.
Monospace bold font	Used to represent commands or other text that the user types.
Monospace italic font	Used to represent variables the reader supplies in command-line examples and terminal sessions.
Shell prompts	The MS-DOS® prompt is used in Windows.
Note	Represents a note.
`Code`	Represents code.

Code Samples

The build volumes, Volume 2: *Migrate Using Windows Service for UNIX 3.5*, Volume 3: *Migrate Using Win32/64,* and Volume 4: *Migrate Using .NET,* of this guide contain several code samples to illustrate certain programming concepts. These code samples are available as source files in a Tools and Templates folder in the download version of this guide, available at http://go.microsoft.com/fwlink/?LinkId=30864.

Chapter 1: Introduction to Win32/Win64

This chapter describes how you can modify the source code of your UNIX applications so that the code compiles on the Microsoft® Windows® operating system using the Microsoft Win32® and Win64 application programming interfaces (APIs). This chapter also describes the improvements that Win64 offers over Win32. It discusses the differences in the Win64 design and the 64-bit UNIX design in terms of Windows and UNIX data models—namely, the P64 of Windows (also called LLP64) and the LP64 of UNIX. This chapter also includes a brief overview of the differences in the way the various resources are managed in UNIX and Windows.

Overview of Win32/Win64

This section gives an overview of the 32-bit and 64-bit APIs, their implementation mechanism on UNIX and Windows, and the differences between them. With this knowledge, you can identify key sections of the application to be modified for compatibility in the Win32/Win64 environment. You can also obtain information on the critical points to consider when porting the UNIX application to the Windows environment using the Win32/Win64 APIs.

UNIX applications are mostly written in C or C++ languages or are based on shell scripts. In both cases, Microsoft provides good support for porting the applications to Windows.

When porting UNIX applications to Windows, the UNIX system calls in the UNIX applications must be mapped to the corresponding Windows API calls. There are also differences in the UNIX architecture and 64-bit Windows (Windows Server™ 2003) architecture, which affect the performance of the UNIX application on Windows.

The following sections discuss the improvements that Win64 offers over Win32 and also compares the 64-bit architecture of Windows to that of UNIX.

Note The Windows API was formerly called the Win32 API. The name Windows API more accurately reflects its roots in 16-bit Windows and its support on 64-bit Windows. The name Windows API is used in this volume except when comparing 32-bit Windows programming with 64-bit Windows programming. They are then referred to as Win32 and Win64 APIs, respectively.

Overview of 64-Bit Windows

The 64-bit versions of Windows operating systems support up to 16 terabytes of random access memory (RAM) whereas 32-bit versions can only support up to 4 gigabytes. Increased RAM support is a major benefit of using a 64-bit operating system because of the following reasons:

- All or part of each application must be replicated for each user, which requires additional memory. Therefore, by having more memory, each application can support more users.
- Increased memory allows more applications to run simultaneously and remain completely resident in the main memory of the system. Applications resident in the main memory provide better performance.
- Data-intensive applications, such as databases and graphics modeling, also benefit from the larger amount of memory because they can store and manipulate large amounts of data more easily and reliably.
- Scientific and high-performance computing applications also benefit from the large amount of memory available in the 64-bit version as they require relatively larger memory for high precision computations.

Overview of the Windows API

The Windows API allows applications to take full advantage of the power of the Windows family of operating systems. Using the Windows API, you can develop applications that run successfully on all versions of Windows, thereby taking advantage of the features and capabilities unique to each version.

The Windows API consists of the following functional categories:

- Base Services
- Common Control Library
- Graphics Device Interface
- Network Services
- User Interface
- Windows Shell

The standard Windows API provides a uniform development environment. The Windows API provides a good interface to access and use kernel or system objects using handles. The Windows API also provides many synchronization and communication mechanisms.

The Windows API also supports programming on 64-bit versions of Windows operating systems. The Win64 API takes advantage of the benefits of the 64-bit Windows architecture. The programming model and API functions are almost the same as in Win32; the main difference between them is the size of the pointers and, consequently, the parameters holding the pointers. In Win64, the size of the pointers is 64 bits; and in Win32, it is 32 bits.

A new set of data types is also defined in Win64 to write cleaner code. In the Win32 API, data types **long** and pointers were of the same size, so data types such as **DWORD** and pointers could be used interchangeably and could also be used to typecast from one to another. The same code in Win64 would lead to errors because in the Win64 API, **long** is 32 bits while pointer is 64 bits.

64-Bit Programming in UNIX and Windows

The key difference between 64-bit programming in UNIX and Windows is the programming model. UNIX uses the LP64 model and Windows uses the LLP64 model.

The LLP64 data model is sometimes described as a 32-bit model with 64-bit addresses. In this model, **int** and **long** are 32-bit types and pointers are 64 bits. Data objects such as structures, which do not contain pointers, are of the same size as on a 32-bit system. A 64-bit data type **longlong** (or int64) is introduced to substitute for a 64-bit type of **int** and **long**.

In the LP64 data model, **long** is a 64-bit type, pointers are 64 bit, and no new data types are introduced. The difference in data types between the two models are listed in Table 1.1.

Table 1.1. Differences Between LLP64 and LP64 Data Types

Data Type	LLP64 (size in bits)	LP64 (size in bits)
char	8	8
short	16	16
int	32	32
long	32	64
longlong (int64)	64	
pointer	64	64

Following are some additional facts about the LLP64 and LP64 models:

- In the data models of both UNIX/64 and 64-bit versions of Windows Server 2003 operating systems, abstract types are defined in terms of basic types. This means that when you use abstract types, you ensure that parameters and structure fields always contain the correctly sized data for 32-bit or 64-bit compilation.
- In the UNIX/64 data model:
  - The size of **int** is 32 bits and the size of **long** and pointers is 64 bits.
  - There are some new explicitly sized types.
  - There are a few new functions.
  - The most scalable and biggest architecture data type is **long**.
- In the Win64 model:
  - The Windows API follows the Uniform Data Model (UDM). UDM proposes to use identically named data types for both the Win32 and Win64 environments. Using this model, you can maintain a single source code development environment for both Win32 and Win64, provided no architecture-specific design features are implemented.
  - The size of **int** and **long** is 32 bits; the size of **int64** (new type) and pointers is 64 bits.

    Abstract types are identical for 32-bit and 64-bit environments, thus simplifying cross-compilation for both of them.
  - There are explicitly sized and scalable data types.
- Some types are "upgraded" and can also be used in Win32 sources.
  - The scalable and biggest architecture integral type depends upon the platform. Microsoft recommends using conditional compilation of either **long** or **int64**; a preprocessor macro can be defined to simplify this complication.
  - Some functions are revised because of polymorphism. Most of the other APIs remain the same.

For more information on data type differences, refer to the “64-bit Programming in UNIX and Windows” section in Chapter 2, “Developing Phase: Process Milestones and Technology Considerations” of this volume.

Comparison of Win32 and Win64

This section compares the major differences between the Win32 and Win64 APIs:

- **Data model**. The Win64 data model is almost the same as that of Win32. However, new data types and pointers have been added. In the ILP32 data model (of Win32), **integer**, **long**, and pointer data types are 32 bits, whereas in the LLP64 (or P64) data model, the pointer data type is 64 bits.

  Note You need to look at the sections of code where **integer**, **long**, and pointer are used interchangeably. Such code would work in Win32 but cause major issues in Win64, and you might face issues when migrating a UNIX 32-bit application to a Windows 64-bit environment.

- **Environment**. The Win64 API environment is almost the same as the Win32 API environment—unlike the major shift from Win16 to Win32. The Win32 and Win64 APIs are now combined and called the Windows API. Using the Windows API, you can compile the same source code to run natively on either 32-bit Windows or 64-bit Windows. To port the application to 64-bit Windows, just recompile the code.

  The Windows header files are modified so that you can use them for both 32-bit and 64-bit code. The new 64-bit types and macros are defined in a new header file, Basetsd.h, which is present in the set of header files included by Windows.h.

  The Basetsd.h file includes the following:

  - New data-type definitions that you can use to make your application word-size independent.
  - Many helper functions for conversion such as:

    ```
    unsigned long HandleToUlong (const void *h)
    long HandleToLong (const void *h)
    void *LongToHandle (const long h)
    unsigned long PtrToUlong (const void *p)
    unsigned int PtrToUint (const void *p)
    ```

- **Data types**. There are three categories of new data types that are supported on 64-bit Windows: fixed-precision data types, pointer-precision types, and specific-precision pointers.

  These new data types are available in the latest 64-bit SDK. There is also a set of functions available to perform conversions in a safe manner so that the same code would run on both the 32-bit and 64-bit platforms.

  When compiling, use the Win64 compiler to display warnings regarding truncation of pointers, invalid type casts, and any other 64-bit–specific issues. After the code is fixed and the compiler no longer shows these warnings, you can use the code safely on both 32-bit and 64-bit platforms.

  For example, the following code can generate the C4311 warning:

  C4311 warning message is 'variable': pointer truncation from 'type' to 'type'. A 64-bit pointer was truncated to a 32-bit int or 32-bit long.

  ```
  buffer = (PUCHAR)srbControl;
  (ULONG)buffer += srbControl->HeaderLength;
  ```

  To correct the code, make the following changes:

  ```
  buffer = (PUCHAR)srbControl;
  (ULONG_PTR)buffer += srbControl->HeaderLength;
  ```

  Additional information about the Win64 compiler is available at

  http://msdn.microsoft.com/.

- **Process interoperability**. Windows-32-On-Windows-64 (WOW64) is the x86 emulator that enables Win32 applications to be run on 64-bit Windows operating systems.

  Processes can load dynamic-link libraries (DLLs) of the same type. For example, a 64-bit application can load a 64-bit DLL but not a 32-bit DLL, whereas out-of-process Component Object Model (COM) servers can interact with both 32-bit and 64-bit clients. Therefore, the DLLs can be wrapped in an out-of-process COM server to allow communication with any kind of application.

Porting from Win32 to Win64

Programming in the Win64 API is the same as in the Win32 API. Apart from a few new data types, a few data types that have changed, and a few new APIs, the rest of the Win32 APIs remain the same. This is because the required functions have been modified internally to gain maximum benefit from the platform, but the interface stays the same.

The data types that have changed from 32-bit to 64-bit need to be handled carefully in order to make the code compatible with Win64. Details of the same are covered in the "Rules for Making Win32 Code 64-bit Compatible" section in Chapter 2, “Developing Phase: Process Milestones and Technology Considerations” of this volume.

Data Types

Following are the three classes of data types available in Win64:

- **Fixed-precision**. Fixed-precision data types are of the same length in both 32-bit and 64-bit Windows operating systems.
- **Pointer-precision**. As the pointer-precision changes (that is, as it becomes 32 bits on 32-bit Windows and 64 bits with 64-bit Windows), these data types reflect the precision accordingly. Therefore, it is safe to cast a pointer to one of these types when you perform pointer arithmetic; if the pointer-precision is 64 bits, then the type is 64 bits. The count types also reflect the maximum size to which a pointer can refer.
- **Specific-precision pointer types**. There are also new pointer types that explicitly size the pointer. Be cautious when using pointers in 64-bit code. If you declare the pointer using a 32-bit type, the operating system creates the pointer by truncating a 64-bit pointer. (All pointers are 64 bits on 64-bit Windows.)

Note Additional information on data types in Win64 is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/win64/win64/the_new_data_types.asp.

Architectural Differences Between UNIX and Windows

This section discusses the architectural differences between UNIX and Windows relating to process and thread management, memory management, and file management. You can use this information to understand the existing implementation mechanism of the preceding aspects in the UNIX application and to identify the best approach to migrate them into the Windows environment using the Win32/Win64 APIs.

Note These topics are discussed in more detail in subsequent chapters of this volume.

Process and Thread Management

Multitasking operating systems, such as Windows and UNIX, must manage and control multiple processes at once. Each process has its own code, data, system resources, and state. Resources include virtual address space, files, and synchronization objects. Threads are a part of a process; each process has one or more threads running on its behalf. Like a process, a thread has resources and a state associated with it. The Windows and UNIX operating systems both provide processes and threads.

The following sections provide more information on how UNIX and Windows manage processes and threads.

Multitasking

UNIX was designed to be a multiprocessing, multiuser system that enables users to create and run many processes at the same time.

Windows has evolved from the MS-DOS® operating system, which does not support preemptive multitasking. Preemptive multitasking is one of the major features of real-time applications. To port the UNIX real-time application to the Windows XP environment, Real-Time Extensions (RTX) for Windows XP can be used. Additional information on Real-Time Extensions for Windows XP is available at

http://msdn.microsoft.com/embedded/getstart/testimonial/xp/xpqa/jaokeefe/default.aspx.

Windows relies heavily on threads for multitasking instead of processes. A thread is a construct that enables parallel processing within a single process. A user can implement multitasking by using multithreading. Creating a new process in Windows is a relatively more resource-consuming operation.

Multiple Users

One key difference between the UNIX and Windows operating systems is the way in which multiple users log on simultaneously to the same computer.

On UNIX, when a user logs on, a shell process is started to service the user's commands. The UNIX operating system keeps track of users and their processes and prevents processes from interfering with one another.

On Windows, when a user logs on interactively, the Win32 subsystem's Graphical Identification and Authentication (GINA) dynamic-link library creates the initial process for that user. This process is known as the user desktop. This desktop is where all user interaction or activity takes place. Only the logged-on user can control the computing environment (sometimes known as the shell). Other users are not able to log on to that computer at the same time. However, if a user uses Terminal Services or Citrix, Windows can operate in a server-centric mode, much as UNIX does. For more information, refer to the "Windows Terminal Services and Citrix" section later in this chapter.

Multithreading

UNIX threads are built upon the Portable Operating System Interface (POSIX) standard—Pthreads. Implementing Pthreads can be user-based or kernel-based depending on the implementation used by the vendor. Most new UNIX kernels are multithreaded to take advantage of Symmetric Multiprocessing (SMP) computers.

Initially, UNIX did not expose threads to programmers. However, POSIX does have user-programmable threads. In fact, POSIX has two different implementations of threads, depending on the POSIX version. Windows applications use threads to take advantage of SMP computers and to maintain interactive capabilities when some threads take a long time to execute. Windows Server 2003 supports preemptive multitasking, which creates the effect of simultaneously executing multiple threads from multiple processes.

In UNIX, in the case of multiple processes running at the same time, small intervals of central processing unit (CPU) cycles are assigned to each process so that, in actuality, only one process runs at any given point of time. Each process is also assigned a priority, and the process with the highest priority gets the CPU first. To prevent high-priority processes from starving other processes, the scheduler may decrease the priority of the currently running process at each clock tick. This enables preemption of the next process with the highest priority to start.

Windows XP uses a quantum-based, preemptive priority scheduling algorithm. The major difference is that threads are scheduled instead of processes. The currently running thread always has the highest priority. Preemption of any thread can happen when a higher priority thread becomes ready, the current thread terminates, or the time quantum for the current thread is exhausted.

Process Hierarchy

When a UNIX-based application creates a new process, the new process becomes a child of the creating process. This process hierarchy is important, and there are system calls for manipulating child processes.

Unlike UNIX, Windows processes do not share a hierarchical relationship. The creating process receives the process handle and the ID of the process created by it so that a hierarchical relationship can be maintained or simulated if the application requires it to do so. However, the operating system treats all processes as belonging to the same generation.

Note Both Windows and UNIX processes inherit the security settings of the creating process by default.

Daemons and Services

In UNIX, a daemon is a process that the system starts to provide a service to other applications. Typically, the daemon does not interact with users. UNIX daemons are started at boot time from **init** or **rc** scripts.

A Windows service is the equivalent of a UNIX daemon. It is a process that provides one or more facilities to client processes. Typically, a service is a long-running, Windows-based application that does not interact with users and consequently does not include a user interface. Services may start when the system boots and they continue running across logon sessions. Services are controlled by the Service Control Manager (SCM). One of the requirements for writing a service is that it must communicate with the SCM to handle starting, stopping, and installing applications. A service runs in user mode with a specific user identity because it runs in a separate process. The security context of that user determines the capabilities of the service. Most services run as the local system account. This account has elevated access rights on the local computer, but it has no privileges on the network domain. If a service needs to access network resources, it must run as a domain user with enough privileges to perform the required tasks. On UNIX, a daemon runs with an appropriate user name for the service that it provides or as the special user who is anonymous.

Summary of Processes and Threads

Table 1.2 lists the differences between Windows and UNIX in terms of processes and threads.

Table 1.2. Windows and UNIX Processes and Threads

Feature	Windows	UNIX
Primary mechanism	Threads	Processes
Processes	Yes	Yes
Threads	Yes	Yes, but different implementations.
Performance	Very good at creating threads.	Very good at creating processes.
Process hierarchy	No, but the information can be collected and acted on by the application itself.	Yes
Security inherited	Yes	Yes (except setuid)

Memory Management

Both UNIX and Windows use virtual memory to extend the memory available to an application beyond the actual physical memory installed on the computer. In UNIX, virtual memory is handled by the kernel; while in Windows, it is handled by an executive service. Virtual memory uses a number of techniques to perform the following tasks:

- Inform the application that additional memory is available.
- Transparently enhance system performance (and therefore application performance) by reading for disk space as efficiently as possible.

Virtual memory uses areas on the disk to extend real memory. In addition, the virtual memory manager moves program and data files from the hard disk into physical memory only when the files are needed. There is no need to consider virtual memory during the migration process because the virtual memory is managed by the operating system and is transparent to applications.

File Management

This section describes the file system characteristics of UNIX and Windows. Table 1.3 lists the basic features of modern file systems.

Table 1.3. File System Features

Feature	Description
File names	User-defined name associated with the physical file, typically 255 characters or more.
Directories	Named folders to store files, usually arranged in a hierarchical, tree-like structure.
Path names	Way of referring to a specific file or directory at a particular place.
Aliases, links, and shortcuts	Methods for pointing one file at another or for giving a file multiple names.
Security	Method of protecting and controlling access to files and directories.
File information	Method of storing the properties of a file, such as creation date, modification time, size, and location on disk.

Both UNIX and Windows support many different types of file system implementations. Some UNIX implementations support Windows file system types, and some products can be used to provide Windows support for some UNIX file system types.

File Names and Path Names

Everything in the file system is either a file or a directory. UNIX and Windows file systems are both hierarchical, and both operating systems support long file names of up to 255 characters. Almost any character is valid in a file name, except for the following:

- Forward slash mark (/) in UNIX.
- Question mark (?), straight quotation mark ("), forward slash mark and backslash (/ and \), greater than and less than (> and <), asterisk (*), vertical bar (|), and colon (:) in Windows.
- Names that conflict with device names.

In UNIX, a single directory, known as the root, is at the top of the hierarchy. You locate all files by specifying a path from the root. The UNIX notation for file paths is a series of directory names separated by a single forward slash mark, followed by the file or directory name. The root directory is named /, so a path begins with /, for example, /etc/passwd. Paths can also be specified as relative to the current working directory (represented as “.”) or the parent of the current working directory (represented as “..”).

UNIX makes no distinction between files on a local hard drive partition, CD-ROM, floppy disk, or networked file system. All the files appear in one tree under the same root. To accomplish this, UNIX uses a process called *mounting*. New file systems (for example, a hard drive partition) are mounted on an empty directory and then appear as part of the file system directory tree. Hence, UNIX treats any external device as another directory by the process of mounting and places it under the root directory.

The Windows file system uses a drive letter abstraction at the user level to distinguish one disk or partition from another. For example, it may assign a drive letter for each partition and for each network drive. As in UNIX, a path in Windows is defined by a series of directories and a file name, but the separator is a backslash. The drive name (for example, C or D) or the Universal Naming Convention (UNC) name (for example, \\SERVER\\SHARE) may also need to be specified. However, Windows can use "." and ".." just as UNIX does.

UNIX File System Features

UNIX treats all files as streams of data with no boundaries or structure. In UNIX, each file in the file system is described by an *inode*. An inode is not the same as a file name. Instead, it refers to the following information about the file:

- Permissions
- Owner
- Type
- Date and time of creation, last access, and modification
- Size
- Pointers to the data blocks allocated to the file

The inode does not contain the name of the file. A directory contains the file names and associated inodes. UNIX can also create hard links and symbolic links, which allow a file to appear in more than one directory with more than one name. In the UNIX file system, devices are also represented by files. Device files are usually found in the /dev directory. For example, you can run a program and ignore all of its output by redirecting the output to the null device, /dev/null. It is also possible to send data directly to a serial port or terminal by using this technique. Some versions of UNIX even expose memory and running processes in this manner (/dev/mem and /dev/proc, respectively).

Applications, not the operating system, handle file structures. This design imparts a simplicity and uniformity to input/output (I/O), but it can cause performance issues for large files or busy systems if not handled carefully.

Windows File System Features

Windows supports three major file systems: FAT16, FAT32, and NTFS.

- **FAT16**. This file system is supported by all Windows operating systems. It is allocated in clusters, the sizes of which are determined by the size of the partition. The larger the partition, the larger the cluster size. This file system is efficient in speed and storage on volumes smaller than 256 MB.
- **FAT32**. This file system is supported by Windows, as well as a number of newer Microsoft operating systems. The major difference between FAT16 and FAT32 is the volume and cluster sizes. This file system can support drives up to 2 terabytes in size. It also uses smaller clusters, resulting in more efficient usage of disk space relative to the larger FAT16 drives.
- **NTFS**. This is the preferred file system for all computers running Windows. It can support drives up to 16 exabytes and also manages disk space efficiently by using smaller clusters. It provides some other advantages, such as automatic data recovery techniques and compression capability on volumes, folders, files, and disk quotas to set the amount of usage allowed by end users.

The file allocation table (FAT) is a list of entries that map to each cluster on the partition. Each version of the FAT file system uses a different size for FAT entries. The FAT16 file system uses 16 bits for each entry while the FAT32 file system uses 32 bits. It also uses a single linked list data structure to index and store file system data. In NTFS, all file attributes are stored as metadata, and it uses B+ trees to index file system data.

Networked File Systems

File systems do not have to be stored on a local drive (for example, a hard disk or CD-ROM). Users and applications can access them over the network from a server or peer computer. To do this, the operating system uses special file systems that work over the network and are called networked file systems.

The standard UNIX network file system is the network file system (NFS). Developed by Sun Microsystems, this technology is licensed to most UNIX vendors and is designed to integrate into the UNIX file system model. An NFS server exports a directory, and an NFS client mounts the exported directory just as it would mount a local file system. To the user, the networked file system appears to be just another part of the directory tree.

UNIX also has an *automount* mechanism. Automount directories are automatically made available when an application attempts to access them. They are then dismounted after a period of inactivity. The automount mechanism reduces the number of network file systems mounted, thereby simplifying administration.

NFS is a client/server implementation. The actions that are executed on the server are minimal. The server does not keep any state associated with the client and all the state data is kept on the client. This method of retaining state ensures that the server can perform quickly and efficiently but places many requirements on the client.

Server Message Block and Common Internet File System

One of the earliest implementations of network resource sharing for the MS-DOS platform was network basic input/output system (NetBIOS). Features in NetBIOS allowed it to accept disk I/O requests and direct them to file shares on other computers.

The protocol used for this was named server message block (SMB). Additions were made to SMB to apply it to the Internet, and now the protocol is known as Common Internet File System (CIFS).

In Windows, the server shares a directory and the client connects to the UNC of that share. Each network drive usually appears with its own drive letter, such as X.

Windows and UNIX Network File System Interoperability

Windows and UNIX can interoperate by using NFS on Windows or CIFS on UNIX. A number of commercial NFS products are available for Windows. For UNIX, in addition to commercial implementations of CIFS, a software option called Samba is widely used. Samba is an alternative to installing NFS client software on Windows-based computers for interoperability with UNIX-based computers. Samba is an open-source, freeware, server-side implementation of a UNIX CIFS server. To provide file and print services, it implements security in the form of authentication and authorization. It also implements NetBIOS-style name resolution and browsing. On Windows, NFS support can be obtained by installing the Windows Services for UNIX product.

Note Additional information on using Windows Services for UNIX NFS gateway to Windows clients is available at

http://support.microsoft.com/kb/324085.

Summary of File System Differences

The preceding sections discussed the architectures of the UNIX and Windows file systems, both of which are hierarchical but differ in many details. Table 1.4 lists the differences between the Windows and UNIX file systems.

Table 1.4. Summary of File Systems Differences

Feature	Windows	UNIX
Overall structure	Hierarchal, multiple trees	Hierarchal, single tree
Drive names	Yes (for example C, D)	No
Mounting partitions	Yes	Yes
Path separator	\	/
Case-sensitive names	No	Yes
Hard links	No	Yes
Symbolic links	No	Yes
Shortcuts	Yes	No
Network file system	SMB	NFS
Device files	No	Yes
Set user ID	No	Yes
Security	Access control lists (ACLs)	Simple bit permissions

Infrastructure Services

This section discusses the differences in the architecture of UNIX and Windows in the following topics:

- Security
- Handles
- Signals, exceptions, and events
- Interprocess communication
- Networking

Security

UNIX and Windows architectures differ in many ways, including security implementation. This section describes some of these security implementation details and differences.

User Authentication

A user can log on to a computer running UNIX by entering a valid user name and password. Some UNIX implementations require optional extra credentials, such as smart cards (for example, with pluggable authentication modules on Solaris and Linux). A UNIX user can be local to the computer or known to a Network Information System (NIS) domain (a group of cooperating computers). In most cases, the NIS database contains little more than the user name, password, and group.

A user can log on to a computer running Windows by entering a valid user name and password. In addition, Windows requires optional credentials such as certificates and smart cards. A Windows user can be local to the computer, be known on a Microsoft Windows NT® domain, or be known in the Active Directory® directory service. The Windows NT domain contains only a user name, password, and user groups. Although Active Directory contains the same information as the Windows NT domain, it may also contain additional information for the user, such as contact information, organizational data, and certificates.

UNIX Security

UNIX uses a simple security model. The operating system applies security by assigning permissions to files. This model works because UNIX uses files to represent devices, memory, and even processes. Security permissions are applied to users or to groups.

In most cases, users are people who log on to the system, but users can be special users such as system services (daemons). In UNIX, each user has a user identifier (UID), which (unlike Windows) does not have to be unique. A user is logged on to the system when a shell process is running that has the UID of that user. Groups are sets of users. A UNIX group has a group identifier (GID). Every process has a UID and a GID associated with it.

Note The credentials that a user supplies when logging on are usually a user name and a password. Some implementations of UNIX support the use of smart cards for interactive logon. Smart cards support cryptography and help secure storage of private keys and certificates, enabling the strong authentication of users.

Security Permissions

When a user logs on to the system by entering a user name and a password, UNIX starts a shell with the UID and GID of that user. From then on, all access to files and other resources is controlled by the permissions assigned to the UID and GID or the process. The UIDs and GIDs are configured in two files: /etc/passwd and /etc/group, respectively.

Each file in the file system has a bitmap that defines its permissions. Read, write, and execute are the permissions that can be granted. These permissions are grouped in three sets: the owner of the file, the group of the owner, and everybody else (world). A full (long) listing for a file shows the file permissions as a group of nine characters that indicate the permissions for owner, group, and world. The characters **r**, **w**, **x**, and **-** are used to indicate read, write, execute, and no permission, respectively. For example, if the owner of a file has all permissions but the group and world have only read permission, the string is as follows:

```
rwxr--r--
```

Note Some UNIX implementations have extended the basic security model to include ACLs similar to those used in Windows. However, ACLs are not implemented consistently across all versions of UNIX.

Effective UID and Effective GID

There are occasions when a process started by a particular user must access resources for which the user does not have permissions. UNIX has a mechanism to handle this situation. Processes can have effective UIDs and GIDs that are different from the UID and GID of the user or the parent process. An effective UID or GID is one that the operating system uses for the duration of the process.

Network Information System

The UNIX operating system was originally designed to run on a server by itself and not on a network, in a manner similar to stand-alone Windows-based computers. When computers can access resources on other computers on a network, synchronization of users (UIDs) and groups (GIDs) across computers becomes a problem. If the numeric identifiers are not properly synchronized, access requests across the network could incorrectly identify the user or group, which would result in security breaches.

The Network Information System (NIS) solves this problem by using a client/server model for processing requests. One computer on a domain is designated the master computer. Computers that serve as backups to the master are known as subordinate computers. All other computers on the domain are clients. When a client application needs to check credentials, the call is forwarded to the master computer instead of being processed locally as it would be on a computer not running NIS. The master looks up the user information in a database file, called a map, and returns the results.

Windows Security

Windows uses a unified security model that protects all objects from unauthorized access. The system maintains security information for the following:

- **Users**. The people who log on to the system, either interactively by entering a set of credentials (typically user name and password) or remotely through the network. The security context of every user is represented by a logon session. Each process that the user starts is associated with the logon session of the user.
- **Objects**. The secured resources that a user can access. For example, files, synchronization objects, and named pipes represent kernel objects.

Figure 1.1 illustrates the Windows security model and the relationship between the process-level access token, the security descriptor of the object, and the discretionary access control list (DACL) for the security descriptor.

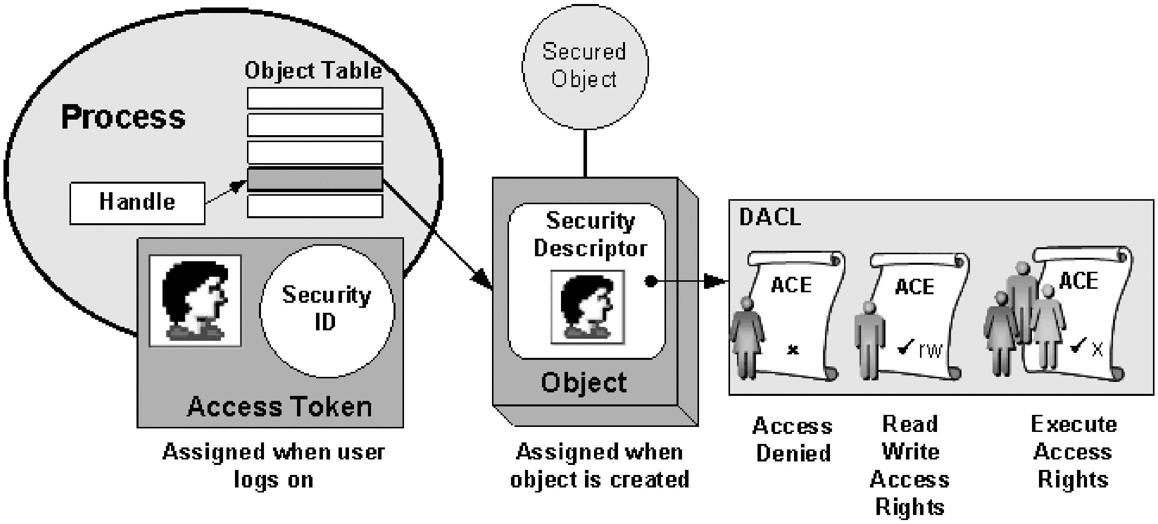


Figure 1.1. The Windows security model

Access Tokens

An access token is a data structure associated with every process that is started by a particular user and is associated with the logon session of that user. The access token identifies the user and identifies security groups that the user is a member of. Although users and groups have human-readable names to ease administration, for performance reasons they are uniquely identified internally by security identifiers (SIDs).

Security Descriptors

A security descriptor describes the security attributes of each object. The information in the security descriptor includes the owner of the object, a system access control list (SACL), and a DACL. The DACL contains a list of access control entries (ACEs) that define the access rights for particular users or groups of users. The owner of the object controls the DACL and uses it to determine who should and should not be allowed access to the object and what rights should be granted to them.

The security descriptor also includes a system access control list (SACL), which is controlled by system administrators. Administrators use SACLs to specify auditing requirements for object access. For example, an administrator can establish a SACL that specifies the generation of an audit log entry whenever a user attempts to delete a particular file.

The sequence of events from the time a user logs on, to the time the user attempts to access a secure object, is as follows:

1. The user logs on by entering a set of credentials. The system validates these credentials by comparing them against the information maintained in a security database (or Active Directory).
2. If the user is authenticated, the system creates a logon session that represents the security context for the user. Every process created on behalf of the user (starting with the Windows shell process) contains an access token that describes the security context of the user.
3. Every process subsequently started by the user is passed a copy of the access token. If one process results in additional processes, all child processes obtain a copy of the access token and are associated with the single logon session of the user.
4. When a process (acting on behalf of the user) attempts to open a secure object, such as a file, the process must initially obtain a handle to the object. For example, when attempting to open a file, the process calls the **CreateFile** function. The process specifies a set of access rights on the call to **CreateFile**.
5. The security system accesses the security descriptor of the object and uses the list of ACEs contained in the DACL to find a group or user SID that matches the one contained in the access token of the process. When this task is complete, the user is either granted a specific set of access rights to the object or denied access to the object (if a deny ACE is located). The granted rights may be the same as the rights initially requested or they may be a subset of the rights initially requested. For example, the **CreateFile** call can request read and write access to a file, but the DACL may allow only read access.

Impersonation

When a thread within a process attempts to access a secured object, the security context that represents the user making the access attempt is normally obtained from the process-level access token.

However, you can associate a temporary access token with a specific thread. For example, within a server process, you can impersonate the security context of a client. The act of impersonation associates a temporary access token with the current thread. The temporary impersonation access token represents the security context of the client. As a result, the server thread uses the security context of the client when it attempts to access any secured object. When the temporary access token is removed from the thread, impersonation ceases and subsequent resource access reverts to using the process-level access token.

Active Directory

Windows Server 2003 introduced the Active Directory directory service, which is used to store information about objects. The objects can include users, computers, printers, and every domain on one or more wide area networks (WANs). Active Directory can scale from a single computer to many large computer networks. Active Directory provides the store for all domain security policy and account information. It replaces the flat account namespace in earlier versions of Windows with a hierarchical namespace for user, group, and computer account information.

Windows Server 2003 includes new authentication protocols based on Internet standards, including Kerberos 5 and Transport Layer Security (TLS). For backward compatibility, Windows Server 2003 supports existing Windows NT LAN Manager Challenge/Response (NTLM) authentication protocols.

The Windows implementation of secure channel security protocols, such as Secure Sockets Layer (SSL) 3.0/TLS, supports strong client authentication by mapping user credentials in the form of public-key certificates to existing Windows NT accounts. Administrators use common administration tools to manage account information and access control, whether the administrators are using password authentication or certificates. External users who do not have Windows Server 2003 accounts can be authenticated through public-key certificates and mapped to an existing Windows account. This allows businesses to give trading partners limited or full access to their internal networks.

Handles

The major differences in implementing system handles in UNIX versus Windows are described in the following sections.

Socket Handles

In UNIX, socket handles are small, non-negative integers. Socket handles can be passed to most of the low-level Portable Operating System Interface (POSIX) input/output (I/O) functions. Windows defines a new unsigned data type SOCKET that may take any value in the range 0 to INVALID_SOCKET–1, where INVALID_SOCKET is a predefined value for a nonexistent socket. Because the SOCKET type is unsigned, compiling existing source code from a UNIX environment may lead to compiler warnings about signed/unsigned data type mismatches.

File Handles

In UNIX, a file handle is an opaque number that is used to uniquely identify a file or other file system object. The only operations that can be carried out with the file handle in UNIX are to copy and compare it for equality with another file handle.

In Windows, the file handle is used to identify a file. When a file is opened by a process using the **CreateFile** function, a file handle is associated with it until either the process terminates or the handle is closed by using the **CloseHandle** function.

Signals, Exceptions, and Events

UNIX and Windows have mechanisms by which processes can indicate an event or error. In both operating systems, these events are signaled by a form of software interrupts. In UNIX, these mechanisms are called signals and are used for ordinary events, such as simple interprocess communication (IPC), and abnormal conditions, such as floating point exceptions. Windows has the following two separate mechanisms:

- An events mechanism handles expected events, such as communication between two processes.
- An exception mechanism handles nonstandard events, such as the termination of a process by the user. Computer hardware may generate exceptions such as invalid memory access and math errors. Windows uses a facility named Structured Exception Handling (SEH) to handle these exceptions.

Interprocess Communication

An operating system designed for multitasking or multiprocessing must provide mechanisms for communicating and sharing data between applications. Such a mechanism is called interprocess communication (IPC). Some forms of IPC are designed for communication among processes running on the same computer, whereas other forms are for communicating across the network between different computers.

UNIX Interprocess Communication

UNIX has several IPC mechanisms that possess different characteristics and are appropriate for different situations. Shared memory, pipes, and message queues are all suitable for processes running on a single computer. Shared memory and message queues are suitable for communicating among unrelated processes. Pipes are the mechanism usually chosen for communicating with a child process through standard input and output. For more information about message queues, refer to the “Message Queues” section later in this chapter.

For communication across the network, sockets are usually the chosen technique. Migration from UNIX sockets to Windows sockets is usually a straightforward process involving few changes to the code.

Windows Interprocess Communication

Windows has many IPC mechanisms, some of which have no counterpart in UNIX. As with UNIX, Windows has shared memory, pipes, and events (equivalent to signals). These are appropriate for processes local to a computer. The shared memory implementation is based on file mapping because certain forms of shared memory can be used across the network. Named pipes can also be used for network communications. Other IPC mechanisms supported by Windows are the Clipboard/Dynamic Data Exchange (DDE), Component Object Model (COM), Distributed Component Object Model (DCOM), and send message. These are mostly used for local communications, but DDE and COM both have network capabilities. Winsock and Message Queuing (also known as MSMQ) are good choices for cross-network tasks.

Windows provides two additional IPC mechanisms: remote procedure call (RPC) and mailslots. RPC is designed for use by client/server applications and is most appropriate for C and C++ programs. Mailslots are memory-based files that a program can access using standard file functions. The maximum size of mailslots is fairly small and their usage is often similar to named pipes except that mailslots are effective for broadcasting small messages.

Synchronization

Both UNIX and Windows have an extensive set of process and thread synchronization techniques. Both operating systems use semaphores, which are synchronization primitives used to control access to a resource that can support a limited number of users. Both UNIX and Windows also use mutex objects to control mutually exclusive access to a resource.

Windows offers critical section objects for lightweight control of multithread access to a section of code. Critical sections are similar to mutexes, but access is limited to the threads of a single process. This makes them appropriate for controlling access to a shared resource. Threads can access the critical section in any order, but the order is not guaranteed.

Message Queues

In UNIX, a message queue is an IPC mechanism. One application sends messages to the queue and another application reads the messages from the queue. The queues are very fast because they are memory based. However, the messages disappear if the system fails. Message queues were introduced in AT&T System V UNIX. Because of this, many versions of UNIX that are based on BSD may not have them. POSIX has message queues, but the API is not exactly the same as in System V.

In recent years, dedicated products have emerged with a more robust and persistent message queue paradigm available on many different platforms. Windows provides a reliable messaging system called Message Queuing. Message Queuing provides guaranteed message delivery, efficient routing, security, and priority-based messaging. In essence, a Message Queuing message is guaranteed to be delivered, but there is no specific guarantee about exactly when it will be received. The operation is the same as on UNIX: One application writes to the queue and another reads from it. The API, however, is completely different.

Shared Memory

As mentioned previously, both Windows and UNIX provide shared memory as one of the IPC mechanisms. In both, the mechanisms provide a section of memory that can be shared between processes to pass data and control information. However, the implementation details are different.

In one of the UNIX implementations, the program must first call a function to get a shared memory identifier, shm_id, for the amount of shared memory. The identifier is then used in calls to attach the shared memory to the process. There are other functions for controlling and removing the shared memory. This type of shared memory mechanism was introduced in the AT&T System v2 version of UNIX. Later UNIX versions introduced shared memory based on the concept of file mapping. The **mmap** function sets up a segment of memory that can be read or written to by two or more programs. This mechanism is used to manipulate files.

The **mmap** function creates a pointer to a region of memory associated with the contents of the file that is accessed through an open file descriptor.

In Windows, implementation of shared memory can be done using the concept of file mapping or by sharing memory using the **GlobalAlloc** function. In the file mapping implementation, a common section of memory can be mapped into the address space of multiple processes. If no file is specified in the creation function, the shared memory is allocated from a section of the page file. As in the UNIX implementation, which uses an identifier, Windows uses a handle identifier to identify the memory that is mapped into the process at the requested address. The **GlobalAlloc** function allocates the specified number of bytes from the heap, which can be shared among processes.

Both the UNIX and Windows file mapping solutions offer the capability of saving the contents in a permanent file.

Pipes

Pipes have similar functionality in both Windows and UNIX systems. Their primary use is to communicate between related processes. UNIX pipes can be named or unnamed. They also have separate read and write file descriptors, which are created through a single function call. With unnamed pipes, a parent process that must communicate with a child process creates a pipe that the child process will inherit and use. Two unrelated processes can use named pipes to communicate with each other.

Windows pipes can also be named or unnamed. A parent process and a child process typically use unnamed pipes to communicate. The processes must create two unnamed pipes for bidirectional communication. Two unrelated processes can use named pipes, even across the network on different computers. Typically, a server process creates the pipe, and clients connect to the bidirectional pipe to communicate with the server process.

Networking

The primary networking protocol for both UNIX and Windows is Transmission Control Protocol/Internet Protocol (TCP/IP). The standard programming API for TCP/IP is called *sockets*. Sockets were created for UNIX at the University of California, Berkeley. Sockets provide an easy-to-use, bidirectional stream between systems across a network. The Windows implementation of sockets is formally known as Windows Sockets but is usually called Winsock. Winsock conforms well to the Berkeley implementation, even at the API level. Most of the functions are the same, but slight differences in parameter lists and return values do exist. It is interesting to note that Winsock allows new transport providers to be installed, making Winsock an extensible environment.

User Interface Differences

The UNIX user interface was originally based on a character-oriented command line, whereas the Windows user interface was originally based on a GUI. This difference is due to the background of the two operating systems. UNIX originated at a time when graphic terminals were not available. Windows was (as the name suggests) designed to take advantage of advances in the graphics capabilities of computers. However, both UNIX and Windows now support a mixture of character and graphical interfaces.

The UNIX Character-based Interface

The standard UNIX shells and tools are all character-based and command-line oriented. For the UNIX shells and UNIX applications to be capable of communicating with different models of character terminals, they must be aware of the different functions available and the command sets for each terminal.

Termcap and Terminfo

To minimize the amount of specific terminal information embedded in a program, UNIX has databases of terminal capabilities. These databases are known as termcap and terminfo. Instead of embedding terminal commands into an application, developers can use program libraries provided with the operating system to query the database for specific movement commands, thus allowing their applications to operate with a variety of hardware.

Curses

Another application development package specifically designed to alleviate the problem of terminal dependence is the curses library originally written at the University of California, Berkeley. Curses is a set of functions used to manipulate terminal input and output (mostly output). These perform actions, such as clearing the screen, moving the cursor to a specific row and column, and writing a character or string to the screen. The library also includes input functions to retrieve user input in various modes, such as read one character and read a string terminated by carriage return.

Curses and similar libraries enable developers to create highly interactive, character-based applications, such as text editors.

X Windows and Motif

The standard windowing system for UNIX systems is the X Window System (or X Windows), developed at the Massachusetts Institute of Technology (MIT). X Windows is a platform-independent, basic windowing system. It consists of a lower-level API called X library (or Xlib) and a higher-level library called X Toolkit Intrinsics. X Windows separates the server (which manages the display of graphical information) from the client (which is the application program that uses X Windows). The server and client can run on separate computers, so the application may run on a powerful numeric server while the output appears on a graphics workstation. This feature has also led to the development of X terminals, that is, computers equipped only to display graphics on a computer screen.

Because X Windows is a set of toolkits and libraries, it does not have graphical user interface standards as Windows does. Motif is the most common windowing system, library, and user interface style built on X Windows. Motif handles Windows and a set of user interface controls known as *widgets*. Widgets cover the whole range of user interface controls, including buttons, scroll bars, menus, and high-functionality items such as a Web browser widget.

X Windows has the capability to support multiple terminals, that is, multiple client sessions can connect to the same X Windows server simultaneously and interact with the server. Windows also has a similar capability known as Windows Terminal Services. This functionality enables users to run multiple interactive sessions on the server at the same time.

Windows Terminal Server and Citrix

Windows can provide sessions that run applications on a server but are displayed on a client workstation. These sessions can be implemented with both Terminal Server (on Windows Server 2003 and Windows XP) and Citrix.

Both Terminal Server and Citrix use a server-based session, much as UNIX does. The difference is that Terminal Server and Citrix use a smart GUI terminal specific to running Windows-based programs. This is analogous to the way an X terminal operates in a UNIX environment. System managers can use Terminal Server to deliver Windows functionality to a low-end computer or even one that does not run Windows. Terminal Server can also be used to remotely administer a Windows-based server.

Terminal Server is particularly useful for implementing server-based applications in a thin client environment. Additionally, Terminal Server provides a smart GUI protocol that works effectively on slow links. This protocol allows enterprises to consolidate applications in a remote location, without the loss of performance usually associated with slower remote networks.

System managers can implement Terminal Server using network load balancing in scale-out server clusters. This configuration allows for both higher availability and the capability to add more servers when the load increases. Applications that use Terminal Server or Citrix usually fall into the following two categories:

- Desktop applications, such as those in the Microsoft Office suite, moved from the desktop client to a central server.
- Remote applications that require thin client connectivity and that are unable to operate through a Web-based interface.

Chapter 2: Developing Phase: Process Milestones and Technology Considerations

This chapter introduces the MSF Developing Phase and helps you prepare for it. It also discusses using Microsoft® Visual Studio® .NET 2003 and the Platform software development kit (SDK) for the Developing Phase. The chapter includes a section about how to make your code compliant with both 32-bit and 64-bit architectures. With this knowledge, you can identify the application development environments, configure the environment, and decide the development and testing methodology to use for the Win32/Win64 applications.

Goals for the Developing Phase

The primary goal during the Developing Phase is to build the solution components—the code migration and the documentation. Typically, the migration involves modifying the existing code in a way that enables the application to work in the Windows environment using the Windows API. In this context, both the modification of existing code and the development of new code are considered to be migration activities. Although the development work is the focus of this phase, all team roles are active in building and testing the deliverables. Also, some development work may continue into the Stabilizing Phase in response to test results.

This phase formally ends with the Scope Complete Milestone. Your team achieves this major milestone by getting a formal approval from the sponsors and key stakeholders. The sponsors and key stakeholders must approve that all solution elements are built and that the solution features and functionality are complete in accordance with the functional specifications developed during the Planning Phase.

Major Tasks and Deliverables

The tasks and deliverables for the Developing Phase are listed in Table 2.1, along with the owners for the task.

Table 2.1. Major Tasks and Deliverables

Major Tasks and Deliverables	Owners
Starting the development cycle The team begins the development cycle by verifying that all tasks identified during the Envisioning and Planning Phases have been completed.	Development
Building a proof of concept Before development starts, the team performs a final verification of the concepts from the designs within an environment that mirrors production as closely as possible.	Development
Developing the solution components The team develops the solution using the core components and extends them to the specific needs of the solution. The team also develops and conducts unit functional tests to ensure that individual features perform according to the specifications.	Development and Test

Major Tasks and Deliverables	Owners
Developing the testing tools and test cases The team develops the testing infrastructure and populates it with test cases. This ensures that the entire solution performs according to specifications. This solution test suite typically incorporates, as a subset, the individual feature tests used by developers to build the solution components.	Test
Building the solution A series of daily or frequent builds culminate with major internal builds and identification of points at which the development team will deliver key features of the solution. These builds are subjected to all or part of the entire project test suite to verify the percentage of completion and are used as a way of tracking the overall progress of the solution and the solution test suite.	Development and Test
Closing the Developing Phase The team completes all features, delivers the code and documentation, and considers the solution complete, thus initiating the approval process for the Scope Complete Milestone.	Project

Note Refer to the *UNIX Migration Project Guide* (UMPG) for an overview of MSF, general information on the processes that belong to each phase, and additional information about the team roles responsible for the processes. The UMPG is meant to be used in conjunction with the technical and solution-specific information in this guide.

Starting the Development Cycle

During the Developing Phase, every component of the solution is analyzed in terms of how to apply code changes to adapt to the Microsoft Windows® environment. This section mainly focuses on identifying and addressing the risks in the Developing Phase and using a mitigation plan to address these risks.

The Developing Phase of any UNIX migration project can be the most challenging part of the project. Major issues become evident at the beginning of this phase. For example, your team may realize that the Windows API does not have an equivalent of a particular UNIX function, which the code is using. This can be categorized as a risk because a replacement code might need to be written at this stage. Resolving such issues will be the distinguishing factor in determining whether schedules will change, whether the funding is sufficient and, ultimately, whether the project will be successful.

If any item on the task lists of the Envisioning and Planning Phases is not completely satisfied, it could present a risk during the Developing Phase.

The following actions might mitigate these risks:

- Procure the required software licenses for Visual Studio .NET 2003 before starting the project.
- Prepare a requirements specification document, which details the scope of the migration project and the design and architecture that must be followed.

- Perform an impact analysis of the changes and obtain sign-off from the customer on the requested changes. Impact analysis plays a very important function during a migration project. The output from this activity helps the developer in determining the scope of the changes, identifying further activities required to fix the problem, and building and testing the changes due to the migration. An impact analysis also helps in identifying the boundaries of migration, identifying the affected elements, and understanding the configuration system for all sources in the application, changes in business process, and types of change with respect to technology changes.
- Establish the existence of compatible versions of required third-party libraries on Windows and procure licenses for the required third-party libraries.

Implementing these actions is easier if the risks are identified and mitigation plans are formulated and evaluated well ahead of time. Risk mitigation, as part of the risk management process, can be used to keep a project on track through adverse situations.

Note Additional information about risk mitigation is available at http://www.microsoft.com/office/solutions/accelerators/sixsigma/default.mspx.

Building a Proof of Concept

Typically, the proof of concept is a continuation of the initial development work (the preliminary proof of concept) that occurred during the Planning Phase. The proof of concept includes developing and testing some key elements of the solution in a nonproduction simulation of the proposed operational environment. The team guides the operations staff and users through the solution to validate their requirements. In addition, during such a review/concept process, the developers may discover design flaws or bugs in the original application being ported that need to be addressed. The proof of concept serves as a "dry run" that tests the worthiness and the ease of the migration process. Pilot migrations help to assess any complications that might occur in the actual migration and also help build confidence in the migration process.

There may be some solution code or documentation that carries through to the eventual solution deliverables. However, the proof of concept is not meant to be production-ready. The proof of concept is considered as throwaway development that gives the team a final chance to verify functional specification content and to address any additional issues before moving into development.

Proof of Concept Complete

The Proof of Concept Complete interim milestone marks the completion of building the proof of concept for the key elements of the solution. The risks associated with the key elements of the solution are identified in this phase, which helps you implement risk management.

Reaching this interim milestone marks the point where the team moves from conceptually validating to building the solution architecture and components.

Developing the Solution Components

The Developing Phase is when the actual solution is built. The individual components are coded and tested to satisfy the project requirements in the Windows environment. Because differences exist between UNIX and Windows, the UNIX code must be modified to work in the Windows environment.

Subsequent chapters of this volume address the potential coding differences related to the following categories:

- Process management
- Thread management
- Memory management
- File management
- Infrastructure services
  - Security
  - Handles
  - Exception handling
  - Signals versus events
  - Interprocess communication
  - Networking
- Migrating Graphical User Interface
- Shells and scripting
- Daemons versus services
- Middleware
- Component-based development in Windows

For each of these categories, chapters 3, 4, 5, and 6 of this volume:

- Describe the coding differences.
- Outline options for converting the code.
- Illustrate the options with source code examples.

You can then choose the solution that is most appropriate to your application and use these instructions as the basis for constructing your Windows code. This guide gives you sufficient information to choose the best method of converting the code. After you have made your choice, you can refer to the MSDN or Microsoft Windows API documentation to ensure that you understand the details of the Microsoft Windows API functions.

Using the Development Environment

The development environment is the environment in which the user develops and builds the solution. The development environment provides the necessary compiler, linker, libraries, and reference objects. In some cases the integrated development environment (IDE) is also provided.

The setting up of the development environment was discussed in general in Chapter 4, “Planning: Setting Up the Development and Test Environments” of Volume 1: *Plan* of this guide. In this volume, application development focuses on Microsoft® Win32®/Win64.

The application may be one of the following:

- A 32-bit application to run on a 32-bit architecture.
- A 64-bit application to run on a 64-bit architecture.
- A 64-bit–ready application that will be executed on a 32-bit architecture and which can be directly ported to the 64-bit architecture just by recompiling.

These three options suggest three alternate ways of deploying the application, but all three have the same code base. The first two are 32-bit and 64-bit applications running on respective architectures. A 64-bit–ready application can be run on both 32-bit and 64-bit computers. The third alternative suggests keeping the application 64-bit–ready on a 32-bit computer. This can be accomplished by using the 64-bit compiler on the 32-bit computer to compile and remove warnings that may come up when the same application is compiled on a 64-bit computer and thereby produce a clean code.

Tools required for developing the solution using the Windows API are:

- Latest Platform SDK.
- Visual Studio .NET 2003.

In Visual Studio .NET 2003, the native 64-bit IDE is still under development, hence no IDE exists for development of 64-bit applications. Therefore, Visual Studio .NET 2003 can be used in conjunction with the compiler for 64-bit applications present in the Platform SDK. The latest Platform SDK includes the compiler, the linker, and other tools for 64-bit development.

The SDK also includes the C-Runtime (CRT) library, the Microsoft Foundation Classes (MFC), and the Active Template Library (ATL) versions for 64-bit production.

Platform SDK

The latest Platform SDK contains tools to build, debug, test, and deliver applications. It also contains all the definitions (include files) and libraries needed to compile programs. In general, SDK tools are run from the command line, just as applications are run from a shell prompt in UNIX. Output from SDK tools can be files created on the disk, text output to the console (such as **stdout** in UNIX), or graphical output to one or more dialog boxes. The Platform SDK also contains definitions and documentation for the Microsoft .NET enterprise servers, including Microsoft BizTalk™ Server, Microsoft Commerce Server, and Microsoft SQL Server™.

The Platform SDK is available on CD or as a free download on the Web. It is also available with the Microsoft Visual C++® development system and Visual Studio .NET 2003, or with an MSDN® Professional or Universal subscription.

Note You can order or download the SDK at
http://www.microsoft.com/downloads/details.aspx?FamilyId=A55B6B43-E24F-4EA3-A93E-40C0EC4F68E5&displaylang=en.

Using the Platform SDK

The Visual Studio .NET 2003 environment can be used as the development environment for development using the Platform SDK. It contains a series of compiler options that can be used to verify or generate various levels of information, as described in the following table.

Table 2.2. Useful Compiler Options

Debug Information Options	**Use**
/Zd	Produces an .obj file or executable file containing only global and external symbols and line-number information, but no symbolic debugging information.
/Z7	Produces an .obj file and an .exe file containing line numbers and full symbolic debugging information for use with the debugger.
/Zi	Creates a program debug database (.pdb) file. The debugger uses this file to step through the source during program execution.
/ZI	Similar to **/Zi**. However, the .pdb file also supports edit and continue.
Additional Debug Information Options	**Use**
/Yd	Places debug information in the .obj files.
/Yl*symbol*	Places arbitrary symbol in the object module.
/Yu,/Yu*filename*	Specifies using a precompiled header (.pch) file during builds.
/YX,/YX*filename*	Instructs the compiler to use a precompiled header file (.pch) if one exists or to create one if none exists.
Runtime Check Options	**Use**
/RTCc	Reports run-time truncation error.
/RTCs	Enables run-time stack checking, includes overrun and under-run of local variables, and enables verification of the stack register.
/RTCu	Enables reporting of variables used without initialization.
/RTC1	Combination of **RTCs** and **RTCu.**

To set these compiler options in the Visual Studio .NET 2003 development environment

1. On the **Project** menu, click **Properties** to open the project's **Property Pages** dialog box.
2. Click the C/C++ folder of **Configuration Properties**.
3. Click **General Property**.
4. Modify the **Debug Information Format** property to set the Debug Information options /Zd, /Z7, /Zi, and /ZI options.
5. Type the compiler option in the **Additional Options** box of **Command Line** property to specify /Yd and /Yl.
6. Click the **Precompiled Headers** property page.
7. Create/Use Precompiled Header property to specify /Yu and /YX options.
8. Click the **Code Generation** property page.

9. Modify the properties **Basic Runtime Checks** or **Smaller Type Check** to specify the Runtime check options.

   Note Additional information on compiler options is available at http://msdn2.microsoft.com/en-us/library/9s7c9wdw(en-US,VS.80).aspx.

Any program written in languages that support the creation of the .pdb files can be used in the Windows debug environment. Thus you can debug an application source file where calls are made using C, C++, and Fortran languages.

You can use the Platform SDK WinDBG tool to open a source file and then run the corresponding debug version of an executable file. WinDBG searches for debug symbol information for the modules that the executable file uses in the symbol image path.

To use WinDBG to debug an executable file

1. From the graphical user interface (GUI), open the source file corresponding to the executable file that you want to debug.
2. Set the path for the symbols needed to debug the executable file.
3. Set desired breakpoints in the source file.
4. Open the executable file.

Other options for using WinDBG include:

- Attach to a running process.
- Open a crash dump file.
- Use a remote debug session to connect to another system.

The command debugger (CDB) can also be used to debug programs. CDB is especially useful for debugging a running program or for debugging remotely. CDB can also analyze crash dumps by using symbols. For example, the following code uses CDB to analyze a crash dump file:

```
cdb –y <Symbol Path> -i <Image Path> -z <Dumpfile>
```

Visual Studio .NET 2003

During the migration of an application from UNIX to Windows, one goal is to take advantage of the Visual Studio development tools. However, it can be a large manual task to manage the creation and administration of Visual Studio projects. This is especially true for large projects with hundreds or thousands of sources. But after the project has been created and the environment is set, the whole application becomes easily manageable.

The Microsoft Visual Studio .NET 2003 IDE can be used to build 64-bit applications while you maintain the same code base for both 32-bit and 64-bit development. You can achieve this by configuring two different 32-bit and 64-bit build environments and compiling the same code base in both environments.

To support developer productivity, Visual Studio .NET 2003 provides debugging and automation facilities, which are particularly useful for repetitive tasks.

Using Visual Studio .NET 2003

The following sections discuss configuring the development environment using the Visual Studio .NET 2003 IDE.

Setting the 64-Bit Build Environment

The process for configuring Visual Studio .NET 2003 to work with the 64-bit Windows is conceptually similar to that for reconfiguring Visual Studio 6, although some of the screens are different.

To set the 64-bit build environment variables

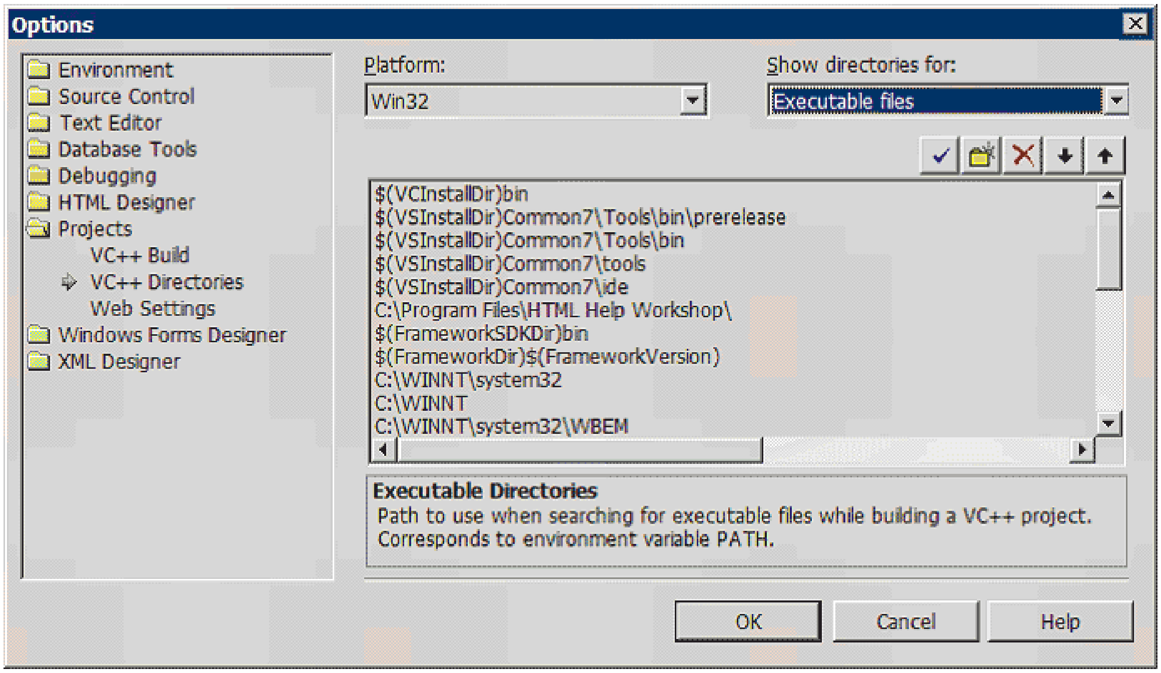
1. In **Programs** of the Microsoft Platform SDK, select **Open Build Environment Window**. Then select **Set Windows XP 64 Build Environment**, and then click **Set Windows XP 64 Build Environment (Debug)**. A console window with the build environment set for a 64-bit build is displayed.
2. At the command prompt:
    a. Change the folder to C:\Program Files\Microsoft SDK.
    b. To load the 64-bit settings in Visual Studio .NET 2003, execute the commands:

       SetEnv.Bat /AMD64 /RETAIL

       devenv /useenv.

       Do not open a new command window to open devenv.exe. The Visual Studio .NET 2003 IDE is displayed, but the include files, the library, and the executable directories are set for a 64-bit build environment.

       Note If devenv.exe is not in the path, change the folder to the \Microsoft Visual Studio .NET 2003\Common7\IDE folder before you run devenv.exe.
3. To resume working with the 32-bit settings, quit Visual Studio .NET 2003 and execute the following statements at the command prompt to relaunch the IDE:
    a. Call "C:\Program Files\Microsoft Visual Studio .NET 2003\VC7\Bin\VCVARS32.BAT
    b. devenv /useenv
4. After you have launched Visual Studio .NET 2003, in the **Tools** menu, click **Options.** Click **Projects** branch, and then **VC++ Directories** to view a sample of the environment, as shown in Figure 2.1.



Platform: Win32

Show directories for: Executable files

```
$(VCInstallDir)bin
$(VSInstallDir)Common7\Tools\bin\prerelease
$(VSInstallDir)Common7\Tools\bin
$(VSInstallDir)Common7\tools
$(VSInstallDir)Common7\ide
C:\Program Files\HTML Help Workshop\
$(FrameworkSDKDir)bin
$(FrameworkDir)$(FrameworkVersion)
C:\WINNT\system32
C:\WINNT
C:\WINNT\system32\WBEM
```

Executable Directories
Path to use when searching for executable files while building a VC++ project. Corresponds to environment variable PATH.

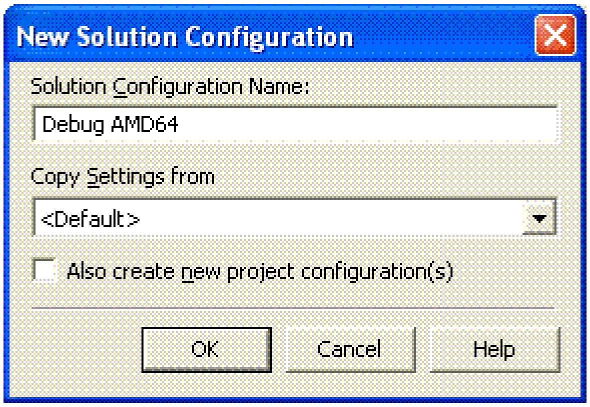
Figure 2.1. Visual Studio .NET 2003 environment

After you have launched Visual Studio .NET 2003 with the proper environment, you must build the right type of release and debug project configurations for the AMD64 platform.

Note Visual Studio .NET 2003 picks up its default tool chain through environment variables defined at startup if you use the /USEENV option.

To add a 64-bit debug configuration

1. On the **Build** menu, click **Configurations**, and then click **Add**.
2. Type a name in the **Configuration** text box, for example, **Release AMD64** or **Debug AMD64**, and from the **Copy settings** list, choose the corresponding configuration for the Win32 platform, such as Win32 Release or Win32 Debug. Clear the **Also create new project configuration(s)** check box.
3. Repeat step 2 for the debug configuration and any other configuration in the project so that for each Win32 configuration, there is a corresponding configuration for the AMD64 platform. Figure 2.2 shows how the configurations will look.



Solution Configuration Name: Debug AMD64

Copy Settings from: <Default>

☐ Also create new project configuration(s)

Figure 2.2 Build configurations

Several Visual Studio compiler and linker options do not apply to the 64-bit compiler and linker.

To modify compiler or linker options

1. On the **Project** menu, click **Properties**.
2. In the **Project Properties** dialog box, click the **General** tab. Under **Output directory** and the **Intermediate Directory** edit boxes, type **Debug64**.
3. On the **C/C++** tab, under **General**, select **Program Database (compiler option, /Zi)** in the **Debug information format** list. Make sure that **Program Database for Edit and Continue** is not enabled.
4. Under Detect 64-bit portability issues, select **Yes (/Wp64)**.
5. On the **C/C++** tab, under **Command Line**, remove **/FD** compiler flag. (The /FD flag is generated when exporting makefiles from earlier versions of Visual Studio and should be deleted.)
6. Remove the **/Gm** compiler flag.
7. Add the **/Wp64** and **/W4** compiler flags to enable the most sensitive IA-64–related compiler warnings.
8. If you want to use 32-bit pointer variables, add the **/Ap32** compiler option (the default is **/Ap64**).
9. If you want to use a maximum of 4 GB (2^32) of virtual address space (Small Address Space), add the **/As32** compiler option (default is **/As64**).
10. In case of an IA64 computer, on the **Linker** tab in **Command Line,** append **/machine:IA64** or **/machine:AMD64** to the options list. Visual Studio .NET 2003 will not let you remove the **/machine:I386** option, but if **/machine:AMD64** comes after it, it can be removed.
11. On the **View** menu, click **Workspace**. To delete the MyApplication.hpj file from the project, click the **MyApplication.hpj** file in the Solution Explorer window, and then press DEL key.

    Note This file may have already been removed.
12. In all 64-bit configurations, under the General branch, change the output directories in order to avoid mixing AMD64 and Win32 object files.

13. If your application is an MFC application, you must add an MFC path to avoid receiving Linkers Tool Error LNK1104 on the Mfc42d.lib file. To add an MFC path, perform the following steps:
    a. On the **Tools** menu, click **Options**.
    b. On the **Projects** tab, select **VC++ Directories.**
    c. On the **Show Directories for** drop-down box, select **Library Files**.
    d. Add the \Microsoft SDK\lib\IA64\mfc path if it is not listed.

    Note If MyApplication is an MFC application and the project uses MFC .dll files, make sure that the .dll files are copied from the \Microsoft SDK\NoRedist\win64 folder to the \System32 folder on the IA64 computer. Following are the DLLs to be copied:

    MFC71.dll.

    This folder also contains the symbols for the MFC, ATL, and MSVCRT debug and release versions.

To build or rebuild the project or solution

1. To build the solution, in Solution Explorer, select or open the desired solution.
2. On the **Build** menu, click **Build Solution** if you want to compile the project files and components that have changed since the last build.
3. On the **Build** menu, click **Rebuild Solution** to clean the solution first, and then build all project files and components.
4. To build any specific project, in Solution Explorer, select or open the specific project.
5. On the **Build** menu, click **Build <project name>** to build only the project files that have changed since the last build.
6. On the **Build** menu, click **Rebuild <project_name>** to clean the project first, and then rebuild all the project files.

After a successful build, you will have a 64-bit application that is ready to be deployed to an IA64 computer.

To debug the .exe file from the Visual Studio .NET 2003 IDE on a 64-bit computer

Note You cannot debug the .exe file from the Visual Studio .NET 2003 IDE.

1. Create a folder named C:\VS2003MSVSMON on the IA64 computer.
2. Copy the following files from the x86 computer to this new folder:
    - Msvcmon.exe
    - Dm.dll
    - Msdis110.dll
    - Tln0t.dll

    These files are located in the \Microsoft Visual Studio .NET 2003\Common7\Packages\Debugger folder.
3. After you copy the files, run Msvcmon.exe on the IA64 computer, and then click **Connect**.
4. In the Visual Studio .NET 2003 IDE on the x86 computer, on the **Build** menu, click **Debugger Remote Connection**. In the **Remote Connection** dialog box, click **Network TCP/IP**, and then click **Settings**. In the **Target computer name or address** box, type the name of the IA64 computer. To close the dialog box, click **OK**.
5. In the Visual Studio C++ IDE, on the **Project** menu, click **Settings**. In the left pane, expand **MyApplication**, and then click the **Debug** tab. You will notice that the **Executable for debug session** textbox contains the path of MyApplication.exe. This will be C:\<X86Path>\MyApplication.exe.
6. In the **Remote executable path and file name** textbox, type **MyApplication.exe** with a full path. This full path looks like \\<X86ComputerName>\C$\<x86Path>\MyApplication.exe. Click **OK** to close the window.
7. To run the .exe file, press CTRL+F5 or click **Execute MyApplication.exe** on the **Build** menu. The .exe file runs on the IA64 computer.

8. If MyApplication is an MFC application and if the project uses MFC .dll files, make sure that the .dll files are copied from the \Microsoft SDK\NoRedist\Win64 folder to the \System32 folder on the IA64 computer. Following are the .dll files:

   - Mfc71.dll

   This folder also contains the symbols for the MFC, the ATL, and the MSVCRT debug and release versions.

Note The 64-bit versions of Microsoft Windows XP and Windows Server™ 2003 include NTSD, a symbolic debugger that works with both 32-bit and 64-bit applications. You can also use the regular WinDbg debugger to let you work on 32-bit applications under 64-bit Windows.

New versions of WinDbg are available for Itanium and native x64 (beta) at

http://www.microsoft.com/whdc/devtools/debugging/64bit-home.mspx.

Debugging with Visual Studio .NET 2003

To debug a program using Visual Studio .NET 2003, the program must be compiled with the appropriate options set. For more information about the options, see Table 2.2 in "Using the Platform SDK" earlier in this chapter. To use the Visual Studio .NET 2003 debugger, in the configuration manager, set the active configuration to debug. When the project is compiled, the debug session can begin.

You can run a debug session within a project opened in Visual Studio .NET 2003 or by opening a remote connection from the IDE. You can choose either option from the **Build** menu in the Visual Studio .NET 2003 IDE. More information on debugging using Visual Studio .NET 2003 is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsdebug/html/_asug_How_Do_I_Topics3a_Debugging.asp.

Note Do not install the 64-bit version of the WinDbg debugging tool on the same computer where Visual Studio .NET 2003 is installed. More information about the 64-bit version of WinDbg is available at the Platform SDK 64-bit Readme.

Additional information is available in the Production Debugging for .NET Framework Applications section at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/DBGrm.asp.

64-Bit Programming in UNIX and Windows

The difference between UNIX/64 and Windows 64-bit programming is because of the different data models. This section describes the new data types in UNIX/64 and Win64 and how they affect your code.

New Explicitly Sized Data Types

Both UNIX/64 and Windows provide new fixed-length or explicitly sized data types. These data types are defined in header files that must be included for either UNIX/64 or Windows Server 2003 (64-bit) programming. In UNIX, they are defined in inttypes.h and in Windows in basetsd.h.

Using explicitly sized types can be helpful in clarifying code purposes and making maintenance easier. However, as these types do not automatically scale with the target architecture, they should not be used for pointers, as pointers in the 64-bit architecture are commonly 64 bits.

UNIX/64 and the Windows UDM have introduced different names for fixed-length types. Fortunately, ANSI standard names, such as **short** and **int**, can still be used for some of these types, which make code migration from UNIX to Windows easier. Whenever possible, it is best to use ANSI types. However, ANSI has no standardized, explicitly sized 64-bit types.

Table 2.3 lists new explicitly sized data types in 64-bit UNIX.

Table 2.3. New UNIX*/64 Explicitly Sized Data Types

Architecture	New Data Types	Old Data Types
64-bit	int64_t, uint64_t	long long, quad_t, u_quad_t
32-bit	int32_t, uint32_t	int, unsigned int
16-bit	int16_t, uint16_t	short, unsigned short
8-bit	int8_t, uint8_t	char

Table 2.4 lists new explicitly sized data types in Win64.

Table 2.4. New Win64 Explicitly Sized Data Types

Architecture	New Data Types	Old Data Types
64-bit	INT64, UINT64, LONG64, ULONG64, DWORD64	__int64, unsigned __int64
32-bit	INT32, UINT32, LONG32, ULONG32, DWORD32	int, unsigned int
16-bit		short, unsigned short
8-bit		char, unsigned char

Notice that with Windows you will no longer be using **int**, **long**, and **dword**. You will need to specify the length if you want to use an explicitly sized type.

New Scalable Data Types

Scalable, or pointer-precision, data types are polymorphic and adjust to the size of the architecture (and thus the pointer size) during compilation. If you use one of these types, it will compile with a 32-bit size under the 32-bit architecture and with a 64-bit size under the 64-bit architecture. You must conditionally compile for these types.

UNIX/64 and Windows Server 2003 (64-bit) have their own respective scalable types. However, there are also some ANSI types available in both environments as listed in Tables 2.5 and 2.6.

Note Do not mix the scalable and fixed data types.

The examples that accompany the tables help you identify how to use the new types. Table 2.5 lists new scalable data types in 64-bit UNIX.

Table 2.5. New UNIX*/64 Scalable Data Types

Description	Data Types
Integral data types that may contain a type-cast pointer.	intptr_t, uintptr_t
Integral data types intended to always contain counting numbers.	long, size_t, ssize_t

Table 2.6 lists new scalable data types in Win64.

Table 2.6. New Win64 Scalable Data Types

Description	Standard	Data Types
Integral data types that may contain a type-cast pointer.	ANSI	intptr_t, uintptr_t
Integral data types that may contain a type-cast pointer.	Win64	LPARAM, WPARAM, LRESULT, INT_PTR, UINT_PTR, WORD_PTR, LONG_PTR, ULONG_PTR
Integral data types intended to always contain counting numbers.	ANSI	size_t, ssize_t
Integral data types intended to always contain counting numbers.	Win64	__int3264, SIZE_T, SSIZE_T

Many Win32 APIs now use these new types, and your code must adapt to them.

There are also new functions in Win64; these are discussed in detail in the following sections.

Rules for Making Win32 Code Compatible with Win64

In general, to ensure that porting occurs comfortably between 32-bit and 64-bit, a few rules must be followed. The following rules can be followed to write code that is compatible with a 64-bit environment, hence having the same code base for 32-bit and 64-bit architectures.

- When using pointers in 64-bit code, if you declare the pointer using a 32-bit type, the operating system creates the pointer by truncating a 64-bit pointer. Pointers are 64 bits on 64-bit Windows. To cast pointers to **int**, **long**, **UINT**, **ULONG**, or **DWORD**, use **UINT_PTR**, **INT_PTR**, **ULONG_PTR**, or **DWORD_PTR**. Make no assumptions about the length of a pointer or xxxx_PTR or xSIZE_T; assume that these are compatible precision.

  Note Do not cast your pointers to the types **ULONG**, **LONG**, **INT**, **UINT**, or **DWORD**.

  HANDLE is defined as a void*, so typecasting a **HANDLE** value to a **ULONG** value to test, set, or clear the low-order 2 bits will cause an error on 64-bit Windows.

- If you must truncate a pointer to a 32-bit value, use the **PtrToLong** or **PtrToUlong** function.

  Note Once truncated, the pointers should never be used as a pointer again or unexpected results, including General Protection Faults, may occur.

- The count types reflect the maximum size to which a pointer can refer. This can be used for a count that must span the full range of a pointer.
- Be cautious when you use unions with pointers.
- Use OUT parameters with caution. Instead, use ULONG_PTR as the data type for OUT parameters.

  Typecasting **&ul** to **PULONG*** prevents a compiler error, but the function will write a 64-bit pointer value into the memory at **&ul**. This code works on 32-bit Windows but will cause data corruption on 64-bit Windows.

- Data structures stored on a disk or exchanged with 32-bit processes need to be handled. Structures that contain the types that change size, for example, **LPARAM**, **WPARAM**, and **LRESULT**, need to be handled.

- Be cautious with polymorphic interfaces. Do not create functions that accept **DWORD** parameters for polymorphic data. If the data can be a pointer or an integral value, use the **UINT_PTR** or **PVOID** type.

  For example, do not create a function that accepts an array of exception parameters typed as **DWORD** values. The array should be an array of **DWORD_PTR** values. Therefore, the array elements can hold addresses or 32-bit integral values. The general rule is that if the original type is **DWORD** and it needs to be pointer width, convert it to a **DWORD_PTR** value. That is why there are corresponding pointer-precision types. If you have code that uses **DWORD**, **ULONG**, or other 32-bit types in a polymorphic way (that is, you want the parameter or structure member to hold an address), use **UINT_PTR** in place of the current type.
- Use the new window-class functions on both 32-bit and 64-bit Windows. Also available is a set of new functions to access pointers or handles in class private data.

  If you have window-class private data that contains pointers, your code must use the following new functions:
  - GetClassLongPtr
  - GetWindowLongPtr
  - SetClassLongPtr
  - SetWindowLongPtr

  Additionally, you must access pointers or handles in class private data using the new functions on 64-bit Windows. To aid you in finding these cases, the following indexes are not defined in Winuser.h during a 64-bit compile:
  - GWL_WNDPROC
  - GWL_HINSTANCE
  - GWL_HWDPARENT
  - GWL_USERDATA

  Instead, Winuser.h defines the following new indexes:
  - GWLP_WNDPROC
  - GWLP_HINSTANCE
  - GWLP_HWNDPARENT
  - GWLP_USERDATA
  - GWLP_ID

  For example, the following code does not compile:

  ```
  SetWindowLong(hWnd, GWL_WNDPROC, (LONG)MyWndProc);
  ```

  It must be changed to the following:

  ```
  SetWindowLongPtr(hWnd, GWLP_WNDPROC, (LONG_PTR)MyWndProc);
  ```
- When setting the **cbWndExtra** member of the **WNDCLASS** structure, be sure to reserve enough space for pointers. For example, if you are currently reserving sizeof (**DWORD**) bytes for a pointer value, reserve sizeof (**DWORD_PTR**) bytes.
- The **LPARAM**, **WPARAM**, and **LRESULT** types change size with the platform. When compiling 64-bit code, these types expand to 64 bits because they typically hold pointers or integral types. Do not mix these values with **DWORD**, **ULONG**, **UINT**, **INT**, **int**, or **long** values. Examine how you use these types and ensure that you do not inadvertently truncate values.

- Access all window and class data using FIELD_OFFSET. Accessing window data using hard-coded offsets is not portable to 64-bit Windows. To make your code portable, access your window and class data using the FIELD_OFFSET macro. Do not assume that the second pointer has an offset of 4.

  ```
  struct foo {
  DWORD NumberOfPointers;
  PVOID Pointers[1];
  } xx;
  Wrong:
  malloc(sizeof(DWORD)+100*sizeof(PVOID));
  Correct:
  malloc(offsetof(struct foo, Pointers)+100*sizeof(PVOID));
  ```
- The **LRESULT** type could be changed to represent a 64-bit value to avoid portability problems.
- Use %p as a format specifier to print pointers.
- Using %x format specifier does not work as expected in a 64-bit environment. Use %I32x or %I64x format specifier instead of %x.
- **HMODULE** is a pointer to the beginning of an EXE or DLL module. In a 64-bit environment, an **HMODULE** must be 64 bits.
- If you pass too few parameters to a function, even if the function is careful not to access that parameter until some other conditions are met, the compiler may find that it needs to discard the parameter, thereby raising the STATUS_REG_NAT_CONSUMPTION exception.
- Assembly code is not x86 (IA32) assembler, so rewrite any assembly code to high-level languages. Also rewrite the 16-bit applications and 16-bit API-based code.
- Be cautious when porting the code that accesses bit fields and bit-wise operations.
- Avoid using hard-coded memory locations.
- Be cautious when porting drivers to 64-bit Microsoft Windows.
- Ensure plug-in interfaces are RPC compliant.
- Enable COM objects to run out-of-process.

Developing the Testing Tools and Test Cases

After developing the solution components, you need to perform testing for the code changes made as part of the development. The testing process helps identify and address potential issues prior to deployment. Testing spans the Developing and the Stabilizing Phases. It starts when you begin developing the solution and ends in the Stabilizing Phase, when the test team certifies that the solution components address the schedule and high-quality goals in the project plan. This also involves using the automated test tools and test scripts.

Figure 2.3 illustrates where the testing activities occur within the phases of the MSF Process Model.

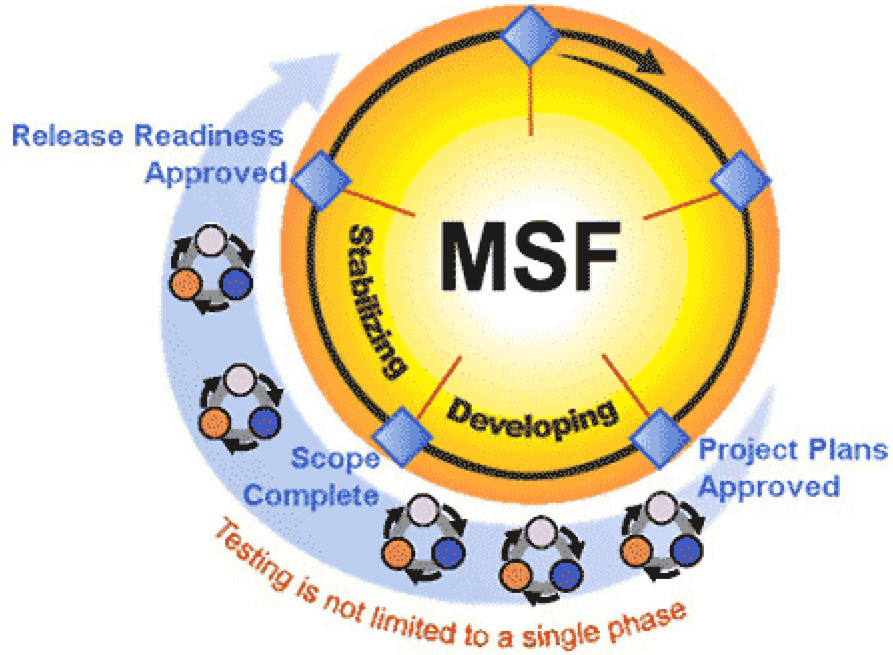

Release Readiness Approved
MSF
Stabilizing
Developing
Scope Complete
Project Plans Approved
Testing is not limited to a single phase

Figure 2.3 MSF Process Model – Testing activities across the Developing and Stabilizing Phases

Testing is performed, parallel to development, throughout the Developing Phase. This section discusses the unit testing activity that needs to be performed during the Developing Phase. The other necessary testing activities are discussed in Chapter 8, “Deployment Considerations and Testing Activities” and Chapter 9, “Stabilizing Phase” of this volume.

During the Developing Phase, testing is not done as a stand-alone activity, but in conjunction with the building of the solution. When building software, the development team designs, documents, and writes the code. Testing is done at this stage through unit testing (discussed in the following section) and daily builds. The testing team designs and documents test specifications and test cases, writes automated scripts, and runs acceptance tests on components submitted for a formal round of testing. The testing team assesses the solution, makes a report on its overall quality and feature completeness, and certifies that the solution features, functions, and components meet the project goals.

Testing in migration projects involving infrastructure services is focused on finding discrepancies between the behavior of the original application, as seen by its clients, and that of the newly migrated application. All discrepancies must be investigated and fixed. It is best to add any new functionality to a migrated application or new capabilities to a migrated service in a separate project initiated after migration is complete.

Unit Testing

Unit testing is the process of verifying whether a specific unit (which can be a class, a program, or a specific functionality) of the code is working according to its functional specifications. It also helps in determining whether the specific unit will be capable of interacting with the other units as defined in the functional specifications.

Unit testing in a UNIX to Windows migration project is the process of finding the discrepancies between the functionality and output of the individual units in the Windows application and the original UNIX application. This might not always be the case; in some cases the design in Windows may differ from the UNIX design, thereby identifying units that are different from the UNIX units. Basic smoke testing, boundary conditions, and error tests are done based on the functional specification of the unit.

The test cases for unit testing include constraints on the inputs and outputs (pre-conditions and post-conditions), the state of the object (in case of a class), the interactions between methods, attributes of the object, and other units.

The unit test cases for migrating UNIX to Win32/64 should mainly focus on the following:

- Data type size validation to identify overflow and truncation errors.
- Parameter data type validation to the APIs.
- Memory allocation routines and usage.
- File size and offset length validations.
- Data type casting.
- Extrapolate test cases on boundary conditions.
- Usage of bit fields and bitwise operations.

Building the Solution

By this stage, the individual components of the solution are developed and tested in the Windows environment using Win32/Win64 APIs to satisfy the project requirements. This stage helps you build the solution with the developed and tested components, and then make the migrated application ready for internal release.

As a good practice, MSF recommends that teams working on development projects perform daily builds of their solution. In migration projects, on the other hand, you typically have to examine large bodies of existing code to understand what they are intended for and to make changes to the code. However, code changes can happen only after addressing porting issues, hence daily builds may not be required. The process of creating interim builds allows a team to find issues early in the development process, which shortens the development cycle and lowers the cost of the project. Note that these interim builds are not deployed in the live production environment. Only when the builds are thoroughly tested and stable are they ready for a limited pilot release to a subset of the production environment. Rigorous configuration management is essential to keeping builds in synch.

Interim Milestone: Internal Release

The project needs interim milestones to help the team measure their progress in the actual building of the solution during the Developing Phase. Each internal release signifies a major step toward the completion of the solution feature sets and achievement of the associated quality level. Depending on the complexity of the solution, any number of internal releases may be required. Each internal release represents a fully functional addition to the solution's core feature set that is potentially ready to move on to the Stabilizing Phase. As each new release of the application is built, fewer bugs must be reported and triaged. Each release marks a significant progress in the approach of the team toward deployment. With each new candidate, the team must focus on maintaining tight control over quality.

The subsequent chapters of this volume describe the necessary code changes required for the migration of the UNIX code to the Windows environment using Win32/Win64 APIs. You can use these instructions to develop the solution components in the Developing Phase.

Chapter 3: Developing Phase: Process and Thread Management

This chapter discusses the similarities and differences in the implementation of process and thread management in UNIX and Microsoft® Windows® operating systems. The chapter first discusses the UNIX and Windows process management mechanism and the Windows application programming interface (APIs) related to processes. It then discusses threads and their implementation in Windows.

Process Management

The UNIX and Windows process management mechanisms are very different, and the major difference between them lies in the creation of processes. UNIX uses **fork** to create a new copy of a running process and **exec** to replace the current process image with the new process image. Windows does not have a **fork** function. Instead, Windows creates processes in one step by using **CreateProcess**. In Windows, there is no need to execute the process after its creation as it will already be executing the new code. However, the standard **exec** functions are still available in Windows. These differences (and others) result in the need to convert the UNIX code before it can run on a Windows platform.

The following sections discuss the following process management topics that need to be considered for migration:

- Creating a New Process
- Replacing a Process Image (exec)
- Retrieving Process Information
- Waiting for a Spawned Process
- Processes vs. Threads
- Managing Process Resource Limits
- Limiting File I/O When Using Windows
- Process Accounting
- Managing and Scheduling Processes

This section also introduces Windows jobs, which allow you to group processes together for management purposes. This functionality is not available in UNIX. With the information provided in this section, you will understand the process management routines in UNIX and Windows. Using this information, you will be able to replace UNIX process routines with the corresponding Windows-compatible routines.

Note There are a number of process management functions in the Windows API. For more information on these functions, refer to the Windows API reference on the Microsoft Developer Network (MSDN®) Web site.

Creating a New Process

In UNIX, you can create a new process by using **fork**. The **fork** function creates a child process that is almost an exact copy of the parent process. The fact that the child is a copy of the parent ensures that the process environment is the same for the child as it is for the parent.

In Windows, the **CreateProcess** function enables the parent process to create an operating environment for a new process. The **CreateProcess** function creates a new process and its primary thread. The environment includes the working directory, window attributes, environment variables, execution priority, and command-line arguments. The new process runs the specified executable file in the security context of the calling process. A handle is returned by the **CreateProcess** function, which enables the parent application to perform operations on the process and its environment while executing. Unlike UNIX, the executable file run by **CreateProcess** is not a copy of the parent process and must be explicitly specified in the call to the **CreateProcess** function. If the calling process is impersonating another user, the new process uses the token for the calling process, not the impersonation token. To run the new process in the security context of the user represented by the impersonation token, use the **CreateProcessAsUser** or **CreateProcessWithLogonW** functions.

An alternative to using **CreateProcess** is to use one of the **spawn** functions that are present in the standard C runtime. There are 16 variations of the **spawn** function. Each **spawn** function creates and executes a new process. Many of these functions have the same functionality as the similarly named **exec** functions in UNIX. The **spawn** functions include an additional argument that permits the new process to replace the current process, suspend the current process until the spawned process terminates, run asynchronously with the calling process, or run simultaneously and detach as a background process. For a UNIX application to change the executable file run in the child process, the child process must explicitly call an **exec** function to overwrite the executable file with a new application. The combination of **fork** and **exec** is similar to, but not the same as, **CreateProcess**. The following example shows a UNIX application that forks to create a child process and then runs the UNIX **ps** command by using **execlp**.

UNIX example: Creating a process using fork and exec

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>

int main()
{
Pid_t pid;
printf("Running ps with fork and execlp\n");
pid = fork();
switch(pid)
{
case -1:
perror("fork failed");
exit(1);
case 0:
if (execlp("ps", NULL) < 0) {
perror("execlp failed");
exit(1);
}
break;
default:
break;
}
printf("Done.\n");
exit(0);
}
(Source File: U_CreateProc-UAMV3C3.01.c)
```

You can port this code to Windows by using the Windows **CreateProcess** function discussed earlier, or by using a **spawn** function from the standard C runtime library. In both cases, the old and new processes run parallel and asynchronously. The following example shows how you can port the previous code using the **CreateProcess** function.

Windows example: Creating a process using CreateProcess

```
#include <Windows.h>
#include <process.h>
#include <stdio.h>

void main()
{
STARTUPINFO si;
PROCESS_INFORMATION pi;
GetStartupInfo(&si);
printf("Running Notepad with CreateProcess\n");
CreateProcess(NULL, "notepad", // Name of app to launch
NULL, // Default process security attributes
NULL, // Default thread security attributes
FALSE, // Don't inherit handles from the parent
0, // Normal priority
NULL, // Use the same environment as the parent
NULL, // Launch in the current directory
&si, // Startup Information
&pi); // Process information stored upon return
printf("Done.\n");
exit(0);
}
```

(Source File: W_CreateProc-UAMV3C3.01.c)

The arguments supported by **CreateProcess** (shown in the preceding example) give you a considerable degree of control over the newly created process. This contrasts with the **spawn** functions on UNIX, which do not provide options to set process priority, security attributes, or the debug status. The **_spawn** function creates and executes a new process on Windows. The following example shows how the same code was ported using the **_spawnlp** function.

Windows example: Creating a process using spawn

```
#include <Windows.h>
#include <process.h>
#include <stdio.h>

void main()
{
printf("Running Notepad with spawnlp\n");
_spawnlp( _P_NOWAIT, "notepad", "notepad", NULL );
printf("Done.\n");
exit(0);
}
```

(Source File: W_CreateProc-UAMV3C3.02.c)

Replacing a Process Image (exec)

Each of the functions in the **exec** family replaces the current process image with a new process image.

In UNIX, the **exec** family of functions replaces the executing process image with that of another process image. The new image is constructed from a regular, executable file called the new process image file. As mentioned previously, a **fork** followed by an **exec** is similar to **CreateProcess**. Windows supports the six POSIX variants of the **exec** function plus two additional ones (**execlpe** and **execvpe**). The function signatures are identical and come as part of the standard C runtime. Porting UNIX code that uses **exec** to Windows is easy to understand. The following is a simple UNIX example showing the use of the **execlp** function.

Note For more information about exec support on Windows, refer to the standard C runtime library documentation that comes with the Microsoft Visual Studio® .NET 2003 development system.

UNIX Example: Replacing a process image using exec

```
#include <unistd.h>
#include <stdio.h>

int main()
{
printf("Running ps with execlp\n");
execlp("ps", "ps", "-l", 0);
printf("Done.\n");
exit(0);
}
```

(Source File: U_ReplaceProc-UAMV3C3.01.c)

The preceding example compiles and runs on Windows with only minor modifications. However, it requires an executable file called ps.exe to be available (one is included with the Interix product). If Interix is not installed, this command can be replaced by any other Windows command. The <unistd.h> include file is not a valid header file when using Windows. To use this example when using Windows, you need to change the header file to <process.h>. Doing so allows you to compile, link, and run this simple application.

Process Information

In Windows, current process information is returned to the parent when a child is created.

The structure of the process information returned is:

```
typedef struct _PROCESS_INFORMATION {
  HANDLE hProcess;
  HANDLE hThread;
  DWORD dwProcessId;
  DWORD dwThreadId;
} PROCESS_INFORMATION;
```

When a process is started, information can be specified for its startup state. This is given in the **_STARTUPINFO** structure. Windows information is contained in this structure. For graphical user interface (GUI) processes, this information affects the first window created by the **CreateWindow** function. For console processes, this information affects the console window if a new console is created for the process. A process can use the **GetStartupInfo** function to retrieve the **_STARTUPINFO** structure specified when the process was created.

Waiting for a Spawned Process

In the preceding section, an example showed how you can create an asynchronous process where the parent and child processes execute simultaneously. No synchronization is performed. This section describes how to modify the previous example to include functionality that enables the parent process to wait for the child process to complete or terminate before continuing. To accomplish this in UNIX, a developer would use one of the **wait** functions to suspend the parent process until the child process terminates. The same semantics are available when using Windows. The functions used are different, but the results are the same. When you view the examples, remember that this is not an exhaustive comparison between the two platforms.

A very simple scenario is described; but if you need to expand the scenario to include waiting for multiple child processes, the example of creating a process using **spawn** does not map adequately because it does not include support for this functionality. In this case, you need to consider the **CreateProcess** approach and **WaitForMultipleObjects**. The **WaitForMultipleObjects** function determines whether the wait criteria were met and accordingly returns a value when any one of the specified objects is in the signaled state or if the time-out interval elapses.

The following example shows how UNIX code that waits for a child process can be migrated to Windows.

UNIX example: Waiting for a spawned process

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
pid_t pid;
int tstat;
printf("Running ps with fork and execlp\n");
pid = fork();
switch(pid)
{
case -1:
perror("fork failed");
exit(1);
case 0:
if (execlp("ps", NULL) < 0) {
perror("execlp failed");
exit(1);
}
break;
default:
break;
}
waitpid(pid, &tstat, 0);
printf("Child process %d terminated with code %d.\n", pid, tstat);
exit(0);
}
```

(Source File: U_WaitSpawn-UAMV3C3.01.c)

Windows example: Waiting for a spawned process using CreateProcess

```
#include <Windows.h>
#include <process.h>
#include <stdio.h>

int main()
{
STARTUPINFO si;
PROCESS_INFORMATION pi;
DWORD tstat;
GetStartupInfo(&si);
printf("Running ps with CreateProcess\n");
CreateProcess(NULL, "ps", // Name of app to launch
NULL, // Default process security attributes
NULL, // Default thread security attributes
FALSE, // Don't inherit handles from the parent
0, // Normal priority
NULL, // Use the same environment as the parent
NULL, // Launch in the current directory
&si, // Startup Information
&pi); // Process information stored upon return
// Suspend our execution until the child has terminated.
WaitForSingleObject(pi.hProcess, INFINITE);
// Obtain the termination code.
GetExitCodeProcess(pi.hProcess, &tstat);
printf("Child process %d terminated with code %d.\n", pi.dwProcessId,
tstat);
exit(0);
}
```

(Source File: W_WaitSpawn-UAMV3C3.01.c)

Windows example: Waiting for a spawned process using _spawnlp

```
#include <Windows.h>
#include <process.h>
#include <stdio.h>

int main()
{
intptr_t pid;
int tstat;
printf("Running ps with spawnlp\n");
pid = _spawnlp( _P_NOWAIT, "ps", "ps", NULL );
// Suspend our execution until the child has terminated.
// obtain termination code upon completion.
_cwait(&tstat, pid, WAIT_CHILD);
printf("Child process %d terminated with code %d.\n", pid, tstat);
exit(0);
}
```

(Source File: W_WaitSpawn-UAMV3C3.02.c)

Processes vs. Threads

In the next example, the UNIX code forks a process, but does not execute a separate run-time image. This creates a separate execution path within the application. When using Windows, this is achieved by using threads instead of processes. If your UNIX application creates separate threads of execution in this manner, you should use the Windows API **CreateThread**. The process of creating threads is discussed in the “Creating a Thread” section of this chapter.

UNIX example: Code with forking executable

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>

int main()
{
pid_t pid;
int n;
printf("fork program started\n");
pid = fork();

switch(pid)
{
case -1:
perror("fork failed");
exit(1);
case 0:
puts("I'm the child");
break;
default:
puts("I'm the parent");
break;
}

exit(0);
}
```

(Source File: U_Fork-UAMV3C3.01.c)

Managing Process Resource Limits

Developers often want to create processes that run with a specific set of resource restrictions. In some cases, they may impose limitations for the purposes of stress testing or forced failure condition testing. In other cases, the limitations may be imposed to restrict runaway processes from using up all the available memory, CPU cycles, or disk space.

In UNIX, the **getrlimit** function retrieves resource limits for a process, the **getrusage** function retrieves current usage, and the **setrlimit** function sets new limits. The common limit names and their meanings are listed in Table 3.1.

Table 3.1. Common Limit Names and Definitions

Limit	Description
RLIMIT_CORE	The maximum size (in bytes) of a core file created by this process. If the core file is larger than RLIMIT_CORE, the write is terminated at this value. If the limit is set to 0, then no core files are created.
RLIMIT_CPU	The maximum time (in seconds) of CPU time a process can use. If the process exceeds this time, the system generates SIGXCPU for the process.
RLIMIT_DATA	Maximum size (in bytes) of a data segment of the process. If the data segment exceeds this value, the functions **brk**, **malloc**, and **sbrk** will fail.
RLIMIT_FSIZE	The maximum size (in bytes) of a file created by a process. If the limit is 0, the process cannot create a file. If a write or truncation call exceeds the limit, further attempts will fail.
RLIMIT_NOFILE	The highest possible value for a file descriptor, plus one. This limits the number of file descriptors a process may allocate. If the number of files being allocated is more than the value of RLIMIT_NOFILE, functions allocating new file descriptors may fail with the error EMFILE.
RLIMIT_STACK	The maximum size (in bytes) of a stack of the process. The stack will not automatically exceed this limit; if a process tries to exceed the limit, the system generates SIGSEGV for the process.
RLIMIT_AS	Maximum size (in bytes) of the total available memory of a process. If this limit is exceeded, the memory functions **brk**, **malloc**, **mmap**, and **sbrk** will fail with *errno* set to ENOMEM, and automatic stack growth will fail as described for RLIMIT_STACK.

Windows uses job objects to set job limits instead of process limits. Unlike in UNIX, Windows job objects do not have file input/output (I/O) source restrictions. If you require file I/O limits in your application, you need to create your own code to handle this.

Windows Job Objects

Windows supports the concept of job objects, which allows you to group one or more processes into a single entity. A job object allows groups of processes to be managed as a unit. Job objects are namable, securable, sharable objects that control attributes of the processes associated with them. Operations performed on the job object affect all processes associated with the job object.

After a job object has been populated with the desired processes, the entire group can be manipulated for various purposes ranging from termination to imposing resource restrictions.

Job objects can be used to implement the concept of UNIX process groups. In Windows, process groups can be implemented by process control, which uses job objects. A process for which you have defined a process-execution rule can be placed into a process group. Process groups use job objects to define rules for the groups. Table 3.2 lists the Windows job objects structures.

Table 3.2. Windows Job Objects Structures

Member	Description	Notes
JOBOBJECT_BASIC_ACCOUNTING_INFORMATION	Contains basic accounting information for a job object.	This structure holds information on total amount of user-mode execution time and kernel-mode execution time, page faults, and the total number of processes associated with the a job for all active and terminated processes.
JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION	Contains basic accounting and I/O accounting information for a job object.	This structure holds information on the basic accounting and the I/O accounting information for the job. It includes information for all processes that have ever been associated with the job, in addition to the information for all processes currently associated with the job.
JOBOBJECT_BASIC_LIMIT_INFORMATION	Contains basic limit information for a job object.	This structure sets various restrictions on the job object such as time limit, working set size, and active process limit. Refer to Table 3.3 for more information.
JOBOBJECT_BASIC_PROCESS_ID_LIST	Contains the process identifier list for a job object.	This structure holds the ProcessIdList, which holds information on process identifiers for the job object.
JOBOBJECT_BASIC_UI_RESTRICTIONS	Contains basic user-interface restrictions for a job object.	This structure holds the UIRestrictionsClass, which restricts the processes associated with the job for creating/switching desktops, changing display settings and system parameters, and accessing global atoms, handles, and reading/writing to Clipboard.
JOBOBJECT_END_OF_JOB_TIME_INFORMATION	Specifies the action the system will perform when the end-of-job time limit is exceeded.	The default termination action is to terminate all processes and set the exit status. Another way is to post a completion packet to the completion port and, when the system clears the end-of-job time limit, processes in the job can continue their execution.
JOBOBJECT_EXTENDED_LIMIT_INFORMATION	Contains basic and extended limit information for a job object.	Contains basic limit information, such as per-process memory limit or per-job memory limit. These are ignored if the corresponding flags are not set in the LimitFlags

Member	Description	Notes
		member of the JOBOBJECT_ BASIC_LIMIT_INFORMATION structure. It also holds peak memory used by processes in the job.
JOBOBJECT_SECURITY_LIMIT_INFORMATION	Contains the security limitations for a job object.	Holds the SecurityLimitFlags, which sets the security limitations for the job and pointers to tokens to specify privileges/control access.

The restrictions that job objects allow you to enforce are in the JOBOBJECT_BASIC_LIMIT_INFORMATION structure.

Table 3.3. Job Objects

Member	Description	Notes
PerProcessUser-TimeLimit	Specifies the maximum user-mode time allotted to each process (in 100 ns intervals).	The system automatically terminates any process that uses more than its allotted time. To set this limit, specify the JOB_OBJECT_ LIMIT_PROCESS_TIME flag in the LimitFlags member.
PerJobUser-TimeLimit	Specifies how much more user-mode time the processes in this job can use (in 100 ns intervals).	By default, the system automatically terminates all processes when the time limit is reached. You can change this value periodically as the job runs. To set this limit, specify the JOB_OBJECT_LIMIT_JOB_TIME flag in the LimitFlags member.
LimitFlags	Specifies the job restrictions to apply.	Refer to the job objects API reference for more information.
MinimumWorkingSetSize/ MaximumWorkingSetSize	Specifies the minimum and maximum working set size for each process (not for all processes within the job).	Normally, the working set of a process can grow beyond its maximum; setting **MaximumWorkingSetSize** forces a hard limit. When the working set of the process reaches this limit, the process pages against itself. Calls to **SetProcessWorkingSetSize** by an individual process are ignored unless the process is just trying to empty its working set. To set this limit, specify the JOB_OBJECT_LIMIT_WORKINGSET flag in the LimitFlags member.
ActiveProcessLimit	Specifies the maximum number of processes that can run concurrently in the job.	Any attempt to go over this limit causes the new process to be terminated with a “not enough quota” error. To set this limit, specify the JOB_OBJECT_LIMIT_ACTIVE_PROCESS flag in the LimitFlags member.
Affinity	Specifies the subset of the CPUs that can run the processes.	Individual processes can limit this even further. To set this limit, specify the JOB_OBJECT_ LIMIT_AFFINITY flag in the LimitFlags member.

Member	Description	Notes
PriorityClass	Specifies the priority class that all processes use.	If a process calls SetPriorityClass, the call will return successfully even though it actually fails. If the process calls GetPriorityClass, the function returns what the process has set the priority class to even though this might not be the actual priority class of the process. In addition, SetThreadPriority fails to raise threads above typical priority but can be used to lower the priority of a process. To set this limit, specify the JOB_OBJECT_LIMIT_PRIORITY_CLASS flag in the LimitFlags member.
SchedulingClass	Specifies a relative time quantum difference assigned to threads in the job.	Value can be from 0 to 9 inclusive; refer to the text after this table for more information. To set this limit, specify the JOB_OBJECT_LIMIT_SCHEDULING_CLASS flag in the LimitFlags member.

As you may have observed by reviewing the table for **setrlimit** and job objects, the restrictions offered by job objects are comparable to UNIX except in one major area—file I/O.

Limiting File I/O When Using Windows

When a process is created in UNIX, the process control block (PCB) in kernel space contains an array of limits that is initialized with default values. In the case of the RLIMIT_FSIZE limit, the write procedures in the kernel are aware of the limit structure in the PCB, and these functions make checks to enforce the limits. The Windows operating system does not implement similar limits on files. To solve this problem, you must write your own solution and build it into your application.

This section presents a solution that you can use in your application. This solution emulates the UNIX file resource limits with:

- An array of limits held as a static variable. This is similar to how some of the C run-time functions use static variables.
- Versions of the UNIX functions **getrlimit()** and **setrlimit()**. These functions manipulate the limit array.
- Wrappers for each of the disk write functions. These wrappers are resource limit-aware.

This solution is implemented as three files. Two of the files, resource.h and resource.c, implement the **getrlimit(), setrlimit(), rfwrite()**, and **_rwrite()** functions. Only **fwrite()** and **_write()** are wrapped because they are the most common disk write functions encountered in the UNIX world. The third file is rlimit.c, which is a very simple test program used to confirm that **rfwrite()** fails when the limit is reached.

Process Accounting

The Windows API has several functions for gathering process accounting information:

- **GetProcessShutdownParameters**. Retrieves shutdown parameters for the currently calling process.
- **GetProcessTimes**. Retrieves timing information for the specified process.
- **GetProcessWorkingSetSize**. Retrieves the minimum and maximum working set sizes of the specified process. **SetPriorityClass**. Sets the priority class for the specified process.

- **SetProcessShutdownParameters**. Sets shutdown parameters for the current calling process.
- **SetProcessWorkingSetSize**. Sets the minimum and maximum working set sizes for the specified process.

Alternatively, a better method of obtaining process information is through the Windows Management Instrumentation (WMI) API.

Note Additional information on WMI is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wmisdk/wmi/wmi_reference.asp.

Managing and Scheduling Processes

This section looks at how you can change the scheduling priority of a process in UNIX and Windows.

In UNIX, **getpriority**(), **setpriority**(), and **nice**() functions can be used to change the priority of processes. The **getpriority**() call returns the current *nice* value for a process, process group, or a user. The returned *nice* value is in the range of [-NZERO, NZERO-1]. NZERO is defined in /usr/include/limits.h. The default process priority always has the value 0 for UNIX. The **setpriority**() call sets the current *nice* value for a process, process group, or a user to the value of *value* + NZERO.

In Windows, processes are scheduled to run based on their scheduling priority. Each thread is assigned a scheduling priority. The priority levels range from zero (lowest priority) to 31 (highest priority). Only the zero-page thread can have a priority of zero. The zero-page thread is a system thread responsible for zeroing any free pages when there are no other threads that need to run.

Each process belongs to one of the priority classes listed in Table 3.4.

Table 3.4. Priority Classes

Priority Classes
IDLE_PRIORITY_CLASS
BELOW_NORMAL_PRIORITY_CLASS
NORMAL_PRIORITY_CLASS
ABOVE_NORMAL_PRIORITY_CLASS
HIGH_PRIORITY_CLASS
REALTIME_PRIORITY_CLASS

By default, the priority class of a process is NORMAL_PRIORITY_CLASS. Use the **CreateProcess** function to specify the priority class of a child process when you create it. If the calling process is IDLE_PRIORITY_CLASS or BELOW_NORMAL_PRIORITY_CLASS, the new process will inherit this class. The **GetPriorityClass** function and the **SetPriorityClass** function can be used to retrieve and set the priority class of processes, respectively.

Table 3.5 lists the functions that are related to scheduling in UNIX and Windows.

Table 3.5. Functions Related to Scheduling in UNIX and Windows

UNIX Function	Description	Windows Function
nice()	Change the priority of a conventional process.	SetPriorityClass
getpriority()	Get the maximum priority of a group of conventional processes.	GetPriorityClass
setpriority()	Set the priority of a group of conventional processes.	SetPriorityClass
sched_getscheduler()	Get the scheduling policy of a process.	GetPriorityClass
sched_setscheduler()	Set the scheduling policy and priority of a process.	SetPriorityClass and SetThreadPriority
sched_getparam()	Get the scheduling priority of a process.	GetThreadPriority
sched_setparam()	Set the priority of a process.	SetThreadPriority
sched_yield()	Relinquish the processor voluntarily without blocking.	For this use SetPriorityClass and set to IDLE_PRIORITY_CLASS
sched_rr_get_interval()	Get the time quantum value for the Round Robin policy.	Not available

For threads, the scheduling priority is determined by the priority class of the process that they belong to and the priority level of the thread. Thread scheduling and priority is discussed in detail in the next section.

Thread Management

This section introduces the concept of threads. The following sections discuss the similarities and differences between UNIX and Windows APIs in managing threads:

- Creating a Thread
- Canceling a Thread
- Synchronization of Threads
- Thread Attributes
- Thread Scheduling and Prioritizing
- Managing Multiple Threads
- I/O Completion Ports

A thread is an independent path of execution in a process that shares the address space, code, and global data of the process. Time slices are allocated to each thread based on priority. Threads consist of an independent set of registers, stack, I/O handles, and message queue.

Threads can usually run on separate processors on multiprocessor computers. Windows enables you to assign threads to a specific processor on a multiprocessor hardware platform.

An application using multiple processes usually has to implement some form of interprocess communication (IPC). This can result in significant overhead and, possibly, a communication bottleneck. In contrast, threads share the process data between them, and interthread communication can be much faster. The problem with threads sharing data is that this can lead to data access conflicts between multiple threads. You can address these conflicts by using synchronization techniques, such as semaphores and mutexes.

In UNIX, threads are implemented by using the POSIX **pthread** functions. In Windows, developers can implement UNIX threading by using the Windows API thread management functions. Although the functionality and operation of threads in UNIX and Windows is very similar, the function calls and syntax are very different.

The following are some similarities between UNIX and Windows in their management of threads:

- Every thread must have an entry point. The name of the entry point is entirely up to you as long as the signature is unique and the linker can adequately resolve any ambiguity.
- Each thread is passed a single parameter when it is created. The contents of this parameter are entirely up to the developer and have no meaning to the operating system.
- A **thread** function must return a value.
- A **thread** function needs to use local parameters and variables as much as possible. When you use global variables or shared resources, threads must use some form of synchronization to avoid potentially corrupting data.

This section discusses how you should go about converting UNIX threaded applications into Windows threaded applications. As a supplement to threads, Windows has a more primitive execution vehicle called a *fiber*. Windows fibers are used to provide full control over scheduling for special needs such as thread pooling, which is useful for certain server applications that manage worker threads for incoming requests.

Note Additional information on the use of fibers is available at http://msdn.microsoft.com/library/en-us/dllproc/base/fibers.asp.

More information on programming with threads in Windows is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore98/HTML/_core_multithreading.3a_.programming_tips.asp.

For details on thread management functions in the Windows API, see the Windows API reference in Visual Studio .NET 2003 or MSDN.

Creating a Thread

When creating a thread in UNIX, use the **pthread_create** function. This function has three arguments: a pointer to a data structure that describes the thread, an argument specifying the attributes of the thread (usually set to NULL, indicating default settings), and the function that the thread will run. The thread finishes execution with a **pthread_exit** where, in this case, it returns a string. The process can wait for the thread to complete using the function **pthread_join**.

The following simple UNIX example creates a thread and waits for it to finish.

UNIX example: Creating a single thread

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

char message[] = "Hello World";

void *thread_function(void *arg)
{
printf("thread_function started. Arg was %s\n", (char *)arg);
sleep(3);
strcpy(message, "Bye!");
pthread_exit("See Ya");
}

int main()
{
int res;
pthread_t a_thread;
void *thread_result;

res = pthread_create(&a_thread, NULL, thread_function, (void
*)message);
if (res != 0)
{
perror("Thread creation failed");
exit (EXIT_FAILURE);
}
printf("Waiting for thread to finish...\n");

res = pthread_join(a_thread, &thread_result);
if (res != 0)
{
perror("Thread join failed");
exit(EXIT_FAILURE);
}

printf("Thread joined, it returned %s\n", (char *)thread_result);
printf("Message is now %s\n", message);
exit(EXIT_SUCCESS);
}
```

(Source File: U_CreateThread-UAMV3C3.01.c)

In Windows, threads are created using the **CreateThread** function, which requires:

- The stack size of the thread.
- The security attributes of the thread.
- The address at which to begin execution of a procedure.
- A pointer to a variable to be passed to the thread.
- Flags that control the creation of the thread.
- An address to store the system-wide unique thread identifier.

After a thread is created, the thread identifier can be used to manage the thread (like get and set the priority of thread) until it has terminated. The next example demonstrates how you should use the **CreateThread** function to create a single thread.

Windows example: Creating a single thread

```
#include <Windows.h>
#include <stdio.h>
#include <stdlib.h>

char message[] = "Hello World";
DWORD WINAPI thread_function(LPVOID arg)
{
printf("thread_function started. Arg was %s\n", (char *)arg);
Sleep(3000);
strcpy(message, "Bye!");
return 100;
}

void main()
{
HANDLE a_thread;
DWORD a_threadId;
DWORD thread_result;

// Create a new thread.
a_thread = CreateThread(NULL, 0, thread_function, (LPVOID)message, 0,
&a_threadId);
if (a_thread == NULL)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}

printf("Waiting for thread to finish...\n");
if (WaitForSingleObject(a_thread, INFINITE) != WAIT_OBJECT_0)
{
perror("Thread join failed");
exit(EXIT_FAILURE);
}

// Retrieve the code returned by the thread.
GetExitCodeThread(a_thread, &thread_result);
printf("Thread joined, it returned %d\n", thread_result);
printf("Message is now %s\n", message);
exit(EXIT_SUCCESS);
}
```

(Source File: W_CreateThread-UAMV3C3.01.c)

Note IA64

1. Do not cast a function that returns void into a LPTHREAD_START_ROUTINE. The kernel raises a STATUS_REG_NAT_CONSUMPTION exception in the IA64 architecture.
2. If you pass too few parameters to a function, even if the function is careful not to access that parameter until some other conditions are met, the compiler may find that it needs to spill the parameter, thereby raising the STATUS_REG_NAT_CONSUMPTION exception in the IA64 architecture.

The UNIX and Windows examples have roughly equivalent semantics. There are only two notable differences:

- The **thread** function in the Windows code cannot return a string value. Developers must use some other means to convey the string message back to the parent (for example, returning an index into a string array).
- The Windows version of the **thread** function just returns a DWORD value instead of calling a function to terminate the thread. **ExitThread** could have been called, but this is not necessary because **ExitThread** is called automatically upon the return from the thread procedure. **TerminateThread** could also be called, but this is neither necessary nor recommended. This is because **TerminateThread** causes the thread to exit unexpectedly. The thread then has no chance to execute any user-mode code, and its initial stack is not deallocated. Furthermore, any DLLs attached to the thread are not notified that the thread is terminating.

Note Additional information on Windows threading routines is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/process_and_thread_functions.asp.

The two solutions have vastly different syntaxes. Windows uses a different set of API calls to manage threads. As a result, the relevant data elements and arguments are considerably different.

Canceling a Thread

The details of terminating threads differ significantly between UNIX and Windows. While both environments allow threads to block termination entirely, UNIX offers additional facilities that allow a thread to specify if it is to be terminated immediately or deferred until it reaches a safe recovery point. Moreover, UNIX provides a facility known as cancellation cleanup handlers, which a thread can push and pop from a stack that is invoked in a last-in-first-out (LIFO) order when the thread is terminated. These cleanup handlers are coded to clean up and restore any resources before the thread is actually terminated. The Windows API allows you to terminate a thread asynchronously. Unlike UNIX, in Windows code you cannot create cleanup handlers and it is not possible for a thread to defer from being terminated. Therefore, it is recommended that you design your code so that threads terminate by returning an exit code and so that threads cannot be terminated forcibly. To do this, you should design your thread code to accept some form of message or event to signal that the threads should be terminated.

Based on this notification, the thread logic can elect to execute cleanup handling code and return normally. To prevent a thread from being terminated, you should remove the security attributes for THREAD_TERMINATE from the thread object. Although forcing a thread to end by using the **TerminateThread** function is not recommended, for completeness, the following example shows how you could convert UNIX code that cancels a thread into Windows code that cancels a thread.

UNIX example: Canceling a thread

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg)
{
int i, res;

res = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
if (res != 0)
{
perror("Thread pthread_setcancelstate failed");
exit(EXIT_FAILURE);
}

res = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
if (res != 0)
{
perror("Thread pthread_setcanceltype failed");
exit(EXIT_FAILURE);
}

printf("thread_function is running\n");
for(i = 0; i < 10; i++)
{
printf("Thread is running (%d)...\n", i);
sleep(1);
}
pthread_exit(0);
}

int main()
{
int res;
pthread_t a_thread;
void *thread_result;

res = pthread_create(&a_thread, NULL, thread_function, NULL);
if (res != 0)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}
sleep(3);
printf("Cancelling thread...\n");

res = pthread_cancel(a_thread);
if (res != 0)
{
perror("Thread cancellation failed");
exit(EXIT_FAILURE);
}

printf("Waiting for thread to finish...\n");
res = pthread_join(a_thread, &thread_result);
if (res != 0)
{
perror("Thread join failed");
exit(EXIT_FAILURE);
}

exit(EXIT_SUCCESS);
}
```

(Source File: U_CancelThread-UAMV3C3.01.c)

Windows example: Cancelling a thread

```
#include <Windows.h>
#include <stdio.h>
#include <stdlib.h>

DWORD WINAPI thread_function(LPVOID arg)
{
printf("thread_function is running. Argument was %s\n", (char *)arg);
for(int i = 0; i < 10; i++)
{
printf("Thread is running (%d)...\n", i);
Sleep(1000);
}
return 100;
}

void main()
{
HANDLE a_thread;
DWORD thread_result;

// Create a new thread.
a_thread = CreateThread(NULL, 0, thread_function, (LPVOID)NULL, 0,
NULL);
if (a_thread == NULL)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}

Sleep(3000);
printf("Cancelling thread...\n");
if (!TerminateThread(a_thread, 0))
{
perror("Thread cancellation failed");
exit(EXIT_FAILURE);
}

printf("Waiting for thread to finish...\n");
WaitForSingleObject(a_thread, INFINITE);
GetExitCodeThread(a_thread, &thread_result);

exit(EXIT_SUCCESS);
}
```

(Source File: W_CancelThread-UAMV3C3.01.c)

When you compare the UNIX and Windows examples, you can see that in the Windows implementation, the setting for the deferred termination is absent. **TerminateThread** is not immediate, and it is not predictable. The termination resulting from a **TerminateThread** call can occur at any point during the thread execution. In contrast, UNIX threads tagged as deferred can terminate when they reach a safe cancellation point.

If you need to match the UNIX behavior in your Windows application exactly, you must create your own cancellation code, thereby preventing the thread from being forcibly terminated.

Synchronization of Threads

UNIX and Windows provide mechanisms for controlling resource access. These mechanisms are referred to as synchronization techniques. In a multithreaded program, you must use synchronization objects whenever there is a possibility of conflict in accessing shared data or resources. For example, if your thread increments a global variable, you cannot predict the result because the variable may have been modified by another thread before or after the increment. The reason that you cannot predict the result is that the order in which threads have access to a shared resource is indeterminate.

The following example illustrates code that is, in principle, indeterminate.

Note This is a very simple example and on most computers the result would always be the same, but the important point to note is that this is not guaranteed.

The main thread in the following example is represented by the parent. It generates a “P,” and the child or secondary thread outputs a “T.” A UNIX example and a Windows example are shown as follows:

UNIX example: Multiple nonsynchronized threads

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg)
{
int count2;
printf("thread_function is running. Argument was: %s\n", (char *)arg);
for (count2 = 0; count2 < 10; count2++)
{
sleep(1);
printf("T");
}
sleep(3);
}
char message[] = "Hello I'm a Thread";

int main()
{
int count1, res;
pthread_t a_thread;
void *thread_result;

res = pthread_create(&a_thread, NULL, thread_function, (void
*)message);
if (res != 0)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}

printf("entering loop\n");
for (count1 = 0; count1 < 10; count1++)
{
sleep(1);
printf("P");
}

printf("\nWaiting for thread to finish...\n");
res = pthread_join(a_thread, &thread_result);
if (res != 0)
{
perror("Thread join failed");
exit(EXIT_FAILURE);
}

printf("\nThread joined\n");
exit(EXIT_SUCCESS);
}
```

(Source File: U_MultiNonSync-UAMV3C3.01.c)

Windows example: Multiple nonsynchronized threads

```
#include <Windows.h>
#include <stdio.h>
#include <stdlib.h>

DWORD WINAPI thread_function(LPVOID arg)
{
int count2;
printf("thread_function is running. Argument was: %s\n", (char *)arg);

for (count2 = 0; count2 < 10; count2++)
{
Sleep(1000);
printf("T");
}

Sleep(3000);
return 0;
}

char message[] = "Hello I'm a Thread";

void main()
{
HANDLE a_thread;
DWORD a_threadId;
DWORD thread_result;
int count1;

// Create a new thread.
a_thread = CreateThread(NULL, 0, thread_function, (LPVOID)message, 0,
&a_threadId);
if (a_thread == NULL)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}

printf("entering loop\n");
for (count1 = 0; count1 < 10; count1++)
{
Sleep(1000);
printf("P");
}

printf("\nWaiting for thread to finish...\n");
if (WaitForSingleObject(a_thread, INFINITE) != WAIT_OBJECT_0)
{
perror("Thread join failed");
exit(EXIT_FAILURE);
}

// Retrieve the code returned by the thread.
GetExitCodeThread(a_thread, &thread_result);
printf("\nThread joined\n");
exit(EXIT_SUCCESS);
}
```

(Source File: W_MultiNonSync-UAMV3C3.01.c)

No actual synchronization between the main thread and child thread is performed; each thread prints different characters. The sequence of the characters printed to the output may be different in each execution.

Once again, it is not possible to predict the output from these examples. In most applications, unpredictable results are an undesirable feature. Consequently, it is important that you take great care in controlling access to shared resources in threaded code.

There are a variety of ways to coordinate multiple threads of execution. To synchronize access to a resource, use one of the synchronization objects in one of the **wait** functions.

The **wait** functions allow a thread to block its own execution. The **wait** functions do not return until the specified criteria have been met. A synchronization object is an object whose handle can be specified in one of the **wait** functions to coordinate the execution of multiple threads. More than one process can have a handle to the same synchronization object, making interprocess synchronization possible.

The next sections discuss the different synchronization techniques.

Synchronization with Interlocked Exchange

A simple form of synchronization is to use what is known as an *interlocked exchange*. An interlocked exchange performs a single operation that cannot be preempted.

The functions **InterlockedExchange**, **InterlockedCompareExchange**, **InterlockedDecrement**, **InterlockedExchangeAdd**, and **InterlockedIncrement** provide a simple mechanism for synchronizing access to a variable that is shared by multiple threads. The threads of different processes can use this mechanism if the variable is in shared memory. (**InterlockedCompareExchange** is discussed in the next section.)

The **InterlockedExchange** function atomically exchanges a pair of values. The function prevents more than one thread from using the same variable simultaneously.

The variable pointed to by the target parameter must be aligned on a 32-bit boundary. These functions fail on multiprocessor x86 systems and any non-x86 systems.

Because this is not the case in the example, the example has limited value; but it does illustrate the use of the **InterlockedExchange** functions.

Note The **InterlockedExchange** function should not be used on memory allocated with the PAGE_NOCACHE modifier.

The following example demonstrates the usage of **InterlockedExchange** for synchronizing the shared resource or global variable.

Windows example: Thread synchronization using interlocked exchange

```
#include <Windows.h>
#include <stdio.h>
#include <stdlib.h>

LONG new_value = 1;
char message[] = "Hello I'm a Thread";

DWORD WINAPI thread_function(PVOID arg)
{
int count2;
printf("thread_function is running. Argument was: %s\n", (char *)arg);

for (count2 = 0; count2 < 10; count2++)
{
Sleep(1000);
printf("(T-%d)", new_value);
InterlockedExchange(&new_value, 1);
}

Sleep(3000);
return 0;
}

void main()
{
HANDLE a_thread;
DWORD a_threadId;
DWORD thread_result;
int count1;

// Create a new thread.
a_thread = CreateThread(NULL, 0, thread_function, (PVOID)message, 0,
&a_threadId);
if (a_thread == NULL)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}

printf("entering loop\n");
for (count1 = 0; count1 < 10; count1++)
{
Sleep(1000);
printf("(P-%d)", new_value);
InterlockedExchange(&new_value, 2);
}

printf("\nWaiting for thread to finish...\n");
if (WaitForSingleObject(a_thread, INFINITE) != WAIT_OBJECT_0)
{
perror("Thread join failed");
exit(EXIT_FAILURE);
}

// Retrieve the code returned by the thread.
GetExitCodeThread(a_thread, &thread_result);
printf("\nThread joined\n");
exit(EXIT_SUCCESS);
}
```

(Source File: W_InterlockEx-UAMV3C3.01.c)

Notes IA64

1. The **InterlockedExchange** function generates a full memory barrier (or fence) and performs the exchange operation. This ensures the strict memory access ordering that is necessary, but it can decrease performance. To operate on 64-bit memory locations and values, use the **InterlockedExchange64** function.

   ```
   LONGLONG InterlockedExchange64(
    LONGLONG volatile* Target,
    LONGLONG Value
   );
   ```
2. The variable pointed to by the *Target* parameter must be aligned on a 64-bit boundary.
3. Be cautious about the variable pointed to by the *Target* parameter and the *Value* parameter; they must be of the data type **longlong**.

Synchronization with Spinlocks

In the previous example, as noted, you still have no synchronization between the two threads. The output may still be out of order. One simple mechanism that offers synchronization is to implement a spinlock. To accomplish this, a variant of the **Interlocked** function called **InterlockedCompareExchange** is used.

The **InterlockedCompareExchange** function performs an atomic comparison of the specified values and exchanges the values, based on the outcome of the comparison. The function prevents more than one thread from using the same variable simultaneously. The **InterlockedCompareExchange** function performs an atomic comparison of the destination value with the comperand value. If the destination value is equal to the comperand value, the exchange value is stored in the address specified by destination, otherwise no operation is performed.

The variables for **InterlockedCompareExchange** must be aligned on a 32-bit boundary; otherwise, this function will fail on multiprocessor x86 systems and any non-x86 systems.

Note This function and all other functions of the **InterlockedExchange** and **InterlockedExchange64** family should not be used on memory allocated with the PAGE_NOCACHE modifier because this may cause hardware faults on some processor architectures. To ensure ordering between reads and writes to PAGE_NOCACHE memory, use explicit memory barriers in your code.

Windows example: Thread synchronization using InterlockedCompareExchange

```
#include <Windows.h>
#include <stdio.h>
#include <stdlib.h>

LONG run_now = 1;
char message[] = "Hello I'm a Thread";

DWORD WINAPI thread_function(LPVOID arg)
{
int count2;
printf("thread_function is running. Argument was: %s\n", (char *)arg);

for (count2 = 0; count2 < 10; count2++)
{
if (InterlockedCompareExchange(&run_now, 1, 2) == 2)
printf("T-2");
```

```
else
Sleep(1000);
}

Sleep(3000);
return 0;
}

void main()
{
HANDLE a_thread;
DWORD a_threadId;
DWORD thread_result;
int count1;

// Create a new thread.
a_thread = CreateThread(NULL, 0, thread_function, (PVOID)message, 0,
&a_threadId);
if (a_thread == NULL)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}

printf("entering loop\n");
for (count1 = 0; count1 < 10; count1++)
{
if (InterlockedCompareExchange(&run_now, 2, 1) == 1)
printf("P-1");
else
Sleep(1000);
}

printf("\nWaiting for thread to finish...\n");
if (WaitForSingleObject(a_thread, INFINITE) != WAIT_OBJECT_0)
{
perror("Thread join failed");
exit(EXIT_FAILURE);
}

// Retrieve the code returned by the thread.
GetExitCodeThread(a_thread, &thread_result);
printf("\nThread joined\n");
```

```
exit(EXIT_SUCCESS);
}
```

(Source File: W_SpinLck-UAMV3C3.01.c)

Spinlocks work well for synchronizing access to a single object, but most applications are not this simple. Moreover, using spinlocks is not the most efficient means for controlling access to a shared resource. Running a While loop in the user mode while waiting for a global value to change wastes CPU cycles unnecessarily. A mechanism is needed that does not waste CPU time while waiting to access a shared resource.

When a thread requires access to a shared resource (for example, a shared memory object), it must either be notified or scheduled to resume execution. To accomplish this, a thread must call an operating system function, passing parameters to it that indicate what the thread is waiting for. If the operating system detects that the resource is available, the function returns and the thread resumes. If the resource is unavailable, the system places the thread in a wait state, making the thread nonschedulable. This prevents the thread from wasting any CPU time. When a thread is waiting, the system permits the exchange of information between the thread and the resource. The operating system tracks the resources that a thread needs and automatically resumes the thread when the resource becomes available. The execution of the thread is synchronized with the availability of the resource. Mechanisms that prevent the thread from wasting CPU time include:

- Mutexes
- Critical sections
- Semaphores

Windows includes all three of these mechanisms, and UNIX provides both semaphores and mutexes. These three mechanisms are described in the following sections.

Synchronization Using Mutexes

A *mutex* is a kernel object that provides a thread with mutually exclusive access to a single resource. The state of a mutex object is set to signaled when it is not owned by any thread, and nonsignaled when it is owned. Only one thread at a time can own a mutex object, whose name comes from the fact that it is useful in coordinating mutually exclusive access to a shared resource.

Any thread of the calling process can specify the mutex-object handle in a call to one of the **wait** functions. The single-object **wait** functions return when the state of the specified object is signaled. When the state of the mutex is signaled, one waiting thread is granted ownership, the state of the mutex changes to nonsignaled, and the **wait** function returns. The owning thread uses the **ReleaseMutex** function to release its ownership.

The next example looks at the use of mutexes to coordinate access to a shared resource and to handshake between two threads.

UNIX example: Thread synchronization using mutexes

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define SHARED_SIZE 1024

char shared_area[SHARED_SIZE];
pthread_mutex_t shared_mutex; /* protects shared_area */

void *thread_function(void *arg)
{
pthread_mutex_lock(&shared_mutex);
while(strncmp("done", shared_area, 4) != 0)
{
printf("You input %d characters\n", strlen(shared_area) -1);
pthread_mutex_unlock(&shared_mutex);
pthread_mutex_lock(&shared_mutex);
}
pthread_mutex_unlock(&shared_mutex);
pthread_exit(0);
}

int main()
{
int res;
pthread_t a_thread;
void *thread_result;

res = pthread_mutex_init(&shared_mutex, NULL);
if (res != 0)
{
perror("Mutex initialization failed");
exit(EXIT_FAILURE);
}

res = pthread_create(&a_thread, NULL, thread_function, NULL);
if (res != 0)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}

pthread_mutex_lock(&shared_mutex);
printf("Input some text. Enter 'done' to finish\n");
while (strncmp("done", shared_area, 4) != 0) {
fgets(shared_area, SHARED_SIZE, stdin);
pthread_mutex_unlock(&shared_mutex);
pthread_mutex_lock(&shared_mutex);
}

pthread_mutex_unlock(&shared_mutex);
printf("\nWaiting for thread to finish...\n");

res = pthread_join(a_thread, &thread_result);
if (res != 0) {
perror("Thread join failed");
exit(EXIT_FAILURE);
}

printf("\nThread joined\n");
pthread_mutex_destroy(&shared_mutex);
exit(EXIT_SUCCESS);
}
```

(Source File: U_Mutex-UAMV3C3.01.c)

Windows example: Thread synchronization using mutexes

```
#include <Windows.h>
#include <stdio.h>
#include <stdlib.h>

#define SHARED_SIZE 1024
char shared_area[SHARED_SIZE];
LPCTSTR lpszMutex = "MUTEX-EXAMPLE";
HANDLE shared_mutex;

DWORD WINAPI thread_function(LPVOID arg)
{
HANDLE hMutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE, lpszMutex);
WaitForSingleObject( hMutex, INFINITE );

while(strncmp("done", shared_area, 4) != 0)
{
printf("You input %d characters\n", strlen(shared_area) -1);
ReleaseMutex(hMutex);
WaitForSingleObject(hMutex, INFINITE);
      }
ReleaseMutex(hMutex);
CloseHandle(hMutex);
return 0;
}

void main()
{
HANDLE a_thread;
DWORD a_threadId;
DWORD thread_result;
// Initialize Semaphore object.

shared_mutex = CreateMutex( NULL, TRUE, lpszMutex );
if (shared_mutex == NULL)
{
perror("Mutex initialization failed");
exit(EXIT_FAILURE);
}

// Create a new thread.
a_thread = CreateThread(NULL, 0, thread_function, (LPVOID)NULL, 0,
&a_threadId);
if (a_thread == NULL) {
perror("Thread creation failed");
exit(EXIT_FAILURE);
}
printf("Input some text. Enter 'done' to finish\n");
WaitForSingleObject(shared_mutex, INFINITE);
while(strncmp("done", shared_area, 4) != 0) {
fgets(shared_area, SHARED_SIZE, stdin);
ReleaseMutex(shared_mutex);
WaitForSingleObject(shared_mutex, INFINITE);
}
ReleaseMutex(shared_mutex);

printf("Waiting for thread to finish...\n");
if (WaitForSingleObject(a_thread, INFINITE) != WAIT_OBJECT_0) {
perror("Thread join failed");
exit(EXIT_FAILURE);
}
```

```
// Retrieve the code returned by the thread.
GetExitCodeThread(a_thread, &thread_result);
CloseHandle(shared_mutex);

printf("Thread joined\n");
exit(EXIT_SUCCESS);
}
```

(Source File: W_Mutex-UAMV3C3.01.c)

Synchronization with Critical Sections

Another mechanism for solving this simple scenario is to use a *critical section*. A critical section is similar to **InterlockedExchange** except that you have the ability to define the logic that takes place as an atomic operation.

Critical section objects provide synchronization similar to that provided by mutex objects, except that critical section objects can be used only by the threads of a single process. Critical section objects provide a slightly faster, more efficient mechanism for mutual-exclusion synchronization (a processor-specific test and set instruction) as compared to event, mutex, and semaphore objects, which can also be used in a single-process application. There is no guarantee about the order in which threads will obtain ownership of the critical section; however, the system will be fair to all threads. Unlike a mutex object, there is no way to tell whether a critical section has been abandoned.

The process is responsible for allocating the memory used by a critical section. Typically, this is done by just declaring a variable of type CRITICAL_SECTION. Before the threads of the process can use it, initialize the critical section and then request ownership of a critical section. If the critical section object is currently owned by another thread, the process waits indefinitely for ownership. In contrast, when a mutex object is used for mutual exclusion, the **wait** functions accept a specified time-out interval. The **TryEnterCriticalSection** function attempts to enter a critical section without blocking the calling thread.

A thread uses the **InitializeCriticalSectionAndSpinCount** or **SetCriticalSectionSpinCount** functions to specify a spin count for the critical section object. On single-processor systems, the spin count is ignored and the critical section spin count is set to 0. On multiprocessor systems, if the critical section is unavailable, the calling thread will spin **dwSpinCount** times before performing a wait operation on a semaphore associated with the critical section. If the critical section becomes free during the spin operation, the calling thread avoids the wait operation.

Any thread of the process can release the system resources that were allocated when the critical section object was initialized. After this function has been called, the critical section object can no longer be used for synchronization.

Windows example: Thread synchronization using critical sections

```
#include <Windows.h>
#include <stdio.h>
#include <stdlib.h>

CRITICAL_SECTION g_cs;
char message[] = "Hello I'm a Thread";

DWORD WINAPI thread_function(LPVOID arg)
{
int count2;
```

```
printf("\nthread_function is running. Argument was: %s\n", (char
*)arg);

for (count2 = 0; count2 < 10; count2++)
{
EnterCriticalSection(&g_cs);
printf("T");
LeaveCriticalSection(&g_cs);
}
Sleep(3000);
return 0;
}

void main()
{
HANDLE a_thread;
DWORD a_threadId;
DWORD thread_result;
int count1;
InitializeCriticalSection(&g_cs);

// Create a new thread.
a_thread = CreateThread(NULL, 0, thread_function, (LPVOID)message, 0,
&a_threadId);
if (a_thread == NULL)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}

printf("entering loop\n");
for (count1 = 0; count1 < 10; count1++)
{
EnterCriticalSection(&g_cs);
printf("P");
LeaveCriticalSection(&g_cs);
}
printf("\nWaiting for thread to finish...\n");

if (WaitForSingleObject(a_thread, INFINITE) != WAIT_OBJECT_0)
{
perror("Thread join failed");
exit(EXIT_FAILURE);
}

// Retrieve the code returned by the thread.
GetExitCodeThread(a_thread, &thread_result);
printf("\nThread joined\n");
DeleteCriticalSection(&g_cs);
exit(EXIT_SUCCESS);
}
```

(Source File: W_CriticalSec-UAMV3C3.01.c)

Synchronization Using Semaphores

A *semaphore object* is a synchronization object that maintains a count between zero and a specified maximum value. The count is decremented each time a thread completes a wait for the semaphore object and incremented each time a thread releases the semaphore. When the count reaches zero, no more threads can successfully wait for the semaphore object state to become signaled. The state of a semaphore is set to signaled when its count is greater than zero, and nonsignaled when its count is zero. The semaphore object is useful in controlling a shared resource that can support a limited number of users.

In the following examples, two threads are created that use a shared memory buffer. Access to the shared memory is synchronized using a semaphore. The primary thread (main) creates a semaphore object and uses this object to handshake with the secondary thread (thread_function). The primary thread instantiates the semaphore in a state that prevents the secondary thread from acquiring the semaphore while it is initiated.

After the user types in text at the console and presses ENTER, the primary thread relinquishes the semaphore. The secondary thread then acquires the semaphore and processes the shared memory area. At this point, the main thread is blocked waiting for the semaphore and will not resume until the secondary thread has relinquished control by calling **ReleaseSemaphore**.

In UNIX, the semaphore object functions of **sem_post** and **sem_wait** are all that are required to perform handshaking. With Windows, you must use a combination of **WaitForSingleObject** and **ReleaseSemaphore** in both the primary and the secondary threads in order to facilitate handshaking. The two solutions are also very different from a syntactic standpoint. The primary difference between their implementations is with the API calls that are used to manage the semaphore objects.

One aspect of **CreateSemaphore** that you need to be aware of is the last argument in its parameter list. This is a string parameter specifying the name of the semaphore. You should not pass a NULL for this parameter. All the kernel objects, including semaphores, are named. All kernel object names are stored in a common namespace except if it is a server running Microsoft Terminal Server, in which case there will also be a namespace for each session. If the namespace is global, one or more unassociated applications could attempt to use the same name for a semaphore. To avoid namespace contention, applications should use some unique naming convention. One solution would be to base your semaphore names on globally unique identifiers (GUIDs).

Terminal Server and Naming Semaphore Objects

As mentioned earlier, Terminal Server has multiple namespaces for kernel objects. There is one global namespace, which is used by kernel objects that are accessible by any and all client sessions and is usually populated by services. Additionally, each client session has its own namespace to prevent namespace collisions between multiple instances of the same application running in different sessions.

In addition to the session and global namespaces, Terminal Server also has a local namespace. By default, the named kernel objects of an application reside in the session namespace. It is possible, however, to override what namespace will be used. This is accomplished by prefixing the name with Global\ or Local\. These prefix names are reserved by Microsoft, are case-sensitive, and are ignored if the computer is not operating as a Terminal Server.

UNIX example: Synchronization using semaphores

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define SHARED_SIZE 1024

char shared_area[SHARED_SIZE];
sem_t bin_sem;

void *thread_function(void *arg) {
sem_wait(&bin_sem);
while(strncmp("done", shared_area, 4) != 0) {
printf("You input %d characters\n", strlen(shared_area) -1);
sem_wait(&bin_sem);
}
pthread_exit(NULL);
}

int main()
{
int res;
pthread_t a_thread;
void *thread_result;

res = sem_init(&bin_sem, 0, 0);
if (res != 0)
{
perror("Semaphore initialization failed");
exit(EXIT_FAILURE);
}
res = pthread_create(&a_thread, NULL, thread_function, NULL);
if (res != 0)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}

printf("Input some text. Enter 'done' to finish\n");
while(strncmp("done", shared_area, 4) != 0)
{
fgets(shared_area, SHARED_SIZE, stdin);
sem_post(&bin_sem);
}
printf("\nWaiting for thread to finish...\n");

res = pthread_join(a_thread, &thread_result);
if (res != 0)
{
perror("Thread join failed");
exit(EXIT_FAILURE);
}
printf("\nThread joined\n");

sem_destroy(&bin_sem);
exit(EXIT_SUCCESS);
}
```

(Source File: U_Semph-UAMV3C3.01.c)

Windows example: Synchronization using semaphores

```
#include <Windows.h>
#include <stdio.h>
#include <stdlib.h>

#define SHARED_SIZE 1024

char shared_area[SHARED_SIZE];
LPCTSTR lpszSemaphore = "SEMAPHORE-EXAMPLE";
HANDLE sem_t;

DWORD WINAPI thread_function(LPVOID arg)
{
LONG dwSemCount;
```

```
HANDLE hSemaphore = OpenSemaphore( SYNCHRONIZE |
SEMAPHORE_MODIFY_STATE, FALSE, lpszSemaphore );

WaitForSingleObject( hSemaphore, INFINITE );

while(strncmp("done", shared_area, 4) != 0)
{
printf("You input %d characters\n", strlen(shared_area) -1);
ReleaseSemaphore(hSemaphore, 1, &dwSemCount);
WaitForSingleObject( hSemaphore, INFINITE );
}

ReleaseSemaphore(hSemaphore, 1, &dwSemCount);
CloseHandle( hSemaphore );
return 0;
}

void main()
{
HANDLE a_thread;
DWORD a_threadId;
DWORD thread_result;
LONG dwSemCount;
// Initialize Semaphore object.
sem_t = CreateSemaphore( NULL, 0, 1, lpszSemaphore );

if (sem_t == NULL)
{
perror("Semaphore initialization failed");
exit(EXIT_FAILURE);
}

// Create a new thread.
a_thread = CreateThread(NULL, 0, thread_function, (LPVOID)NULL, 0,
&a_threadId);

if (a_thread == NULL)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}
WaitForSingleObject(sem_t, INFINITE);
printf("Input some text. Enter 'done' to finish\n");
```

```
while(strncmp("done", shared_area, 4) != 0) {
fgets(shared_area, SHARED_SIZE, stdin);
ReleaseSemaphore(sem_t, 1, &dwSemCount);
WaitForSingleObject(sem_t, INFINITE);
}

printf("\nWaiting for thread to finish...\n");
if (WaitForSingleObject(a_thread, INFINITE) != WAIT_OBJECT_0) {
perror("Thread join failed");
exit(EXIT_FAILURE);
}

// Retrieve the code returned by the thread.
GetExitCodeThread(a_thread, &thread_result);
printf("\nThread joined\n");
exit(EXIT_SUCCESS);
}
```

(Source File: W_Semph-UAMV3C3.01.c)

Thread Attributes

There are a number of attributes associated with threads in UNIX that you need to convert to equivalent attributes in Windows. This section contrasts the UNIX and Windows thread attributes and describes how you should convert your code. Table 3.6 lists the relevant UNIX thread attributes.

Table 3.6. UNIX Thread Attributes

Attribute	Default values	Description
detachstate	PTHREAD_CREATE_JOINABLE PTHREAD_CREATE_DETACHED	Thread may be joined by other threads. Threads may not be waited on for termination.
inheritsched	PTHREAD_INHERIT_SCHED PTHREAD_EXPLICIT_SCHED	Scheduling parameters, policy, and scope are inherited from creating thread. Scheduling parameters for the newly created thread are specified in the thread attribute.
schedparam	-	Priority set to default for scheduling policy.
schedpolicy	SCHED_OTHER SCHED_FIFO SCHED_RR	Scheduling policy is determined by the system. Threads are scheduled in a first-in-first-out order. Threads are scheduled in a round-robin fashion.
Scope	PTHREAD_SCOPE_SYSTEM PTHREAD_SCOPE_PROCESS	Threads are scheduled system-wide. Threads are scheduled based on other threads in the owning process.

Attribute	Default values	Description
Stackaddr	N/A	Attribute not supported; address selected by the operating system.
Stacksize	0	Stack size inherited from process stack size attribute.

Detachstate indicates whether a thread can be waited on for termination. In Windows, the same effect is achieved by closing any handles that exist for a given thread. Because a handle is required for one of the wait and thread management functions, without a handle, you are effectively stopped from acting on a thread. You can also control thread objects based on a security descriptor that is provided at the time the thread is created.

Note Additional information on access control is available at http://msdn.microsoft.com/library/en-us/security/Security/access_control.asp.

The handle returned by the **CreateThread** function has THREAD_ALL_ACCESS access to the thread object. When you call the **GetCurrentThread** function, the system returns a pseudohandle with the maximum access that the security descriptor of the thread allows the caller.

The valid access rights for thread objects include the DELETE, READ_CONTROL, SYNCHRONIZE, WRITE_DAC, and WRITE_OWNER standard access rights, in addition to the thread-specific access rights listed in Table 3.7.

Table 3.7. Thread-Specific Access Rights

Value	Meaning
SYNCHRONIZE	A standard right required to wait for the thread to exit.
THREAD_ALL_ACCESS	Specifies all possible access rights for a thread object.
THREAD_DIRECT_IMPERSONATION	Required for a server thread that impersonates a client.
THREAD_GET_CONTEXT	Required to read the context of a thread by using GetThreadContext.
THREAD_IMPERSONATE	Required to directly use the security information of a thread without calling it using a communication mechanism that provides impersonation services.
THREAD_QUERY_INFORMATION	Required to read certain information from the thread object.
THREAD_SET_CONTEXT	Required to write the context of a thread.
THREAD_SET_INFORMATION	Required to set certain information in the thread object.
THREAD_SET_THREAD_TOKEN	Required to set the impersonation token for a thread.
THREAD_SUSPEND_RESUME	Required to suspend or resume a thread.
THREAD_TERMINATE	Required to terminate a thread.

Inheritsched/schedparam/schedpolicy/scope indicates that the scheduling is either inherited from the thread that created the new thread or that it is set explicitly. It also defines the policy and scope applied to scheduling threads. In Windows, by default, the priority of a thread is THREAD_PRIORITY_NORMAL.

You can use the **GetThreadPriority** function to determine the current priority of a thread and the **SetThreadPriority** function to change the priority of a thread.

Stacksize indicates the stack size applied to a thread at the time of its creation by using the **CreateThread** function. The initial size of the stack is specified in bytes. The system rounds this value to the nearest page. If this parameter has a zero value, the new thread uses the default stack size for the executable.

Setting Thread Attributes

This section presents a simple example of how the attributes of a thread can be set. The UNIX example makes some basic use of thread attributes. The corresponding Windows example does not need to use attributes to accomplish the same functionality. All that is required with Windows is to create a thread that cannot be acted upon by a wait. This can be accomplished by passing NULL as the **dwThreadId** parameter to the **CreateThread** function and by closing the handle that is returned by the call.

The net effect of these combined activities effectively hinders the capability of an application to manage the thread. This issue is addressed in the "Thread Scheduling and Prioritizing" section later in this chapter.

UNIX example: Setting thread attributes

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

char message[] = "Hello I'm a Thread";
int thread_finished = 0;

void *thread_function(void *arg)
{
printf("thread_function running. Arg was: %s\n", (char *)arg);
sleep(4);
printf("Second thread setting finished flag, and exiting now\n");
thread_finished = 1;
pthread_exit(NULL);
}

int main()
{
int count=0, res;
pthread_t a_thread;
void *thread_result;
pthread_attr_t thread_attr;

res = pthread_attr_init(&thread_attr);
if (res != 0) {
perror("Attribute creation failed");
exit(EXIT_FAILURE);
}

res = pthread_attr_setdetachstate(&thread_attr,
PTHREAD_CREATE_DETACHED);
if (res != 0) {
perror("Setting detached attribute failed");
exit(EXIT_FAILURE);
}

res = pthread_create(&a_thread, &thread_attr, thread_function, (void
*)message);
if (res != 0)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}

(void)pthread_attr_destroy(&thread_attr);
while(!thread_finished) {
printf("Waiting for thread to finish (%d)\n", ++count);
sleep(1);
}

printf("Other thread finished, See Ya!\n");
exit(EXIT_SUCCESS);
}
```

(Source File: U_ThreadAttr-UAMV3C3.01.c)

Windows example: Setting thread attributes

```
#include <Windows.h>
#include <stdio.h>
#include <stdlib.h>

char message[] = "Hello I'm a Thread";
int thread_finished = 0;

DWORD WINAPI thread_function(LPVOID arg)
{
printf("\nthread_function running. Arg was: %s\n", (char *)arg);
Sleep(4000);
printf("Second thread setting finished flag, and exiting now\n");
thread_finished = 1;
return 100;
}

void main()
{
int count=0;
HANDLE a_thread;
// Create a new thread.
a_thread = CreateThread(NULL, 0, thread_function, (PVOID)message, 0,
NULL);

if (a_thread == NULL)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}
CloseHandle(a_thread);

while(!thread_finished)
{
printf("Waiting for thread to finish (%d)\n", ++count);
Sleep(1000);
}

printf("Other thread finished, See Ya!\n");
exit(EXIT_SUCCESS);
}
```

(Source File: W_ThreadAttr-UAMV3C3.01.c)

Windows Security and Thread Objects

Threads are kernel objects that are protected by Windows security; therefore, a process must request permission before attempts are made to manipulate an object. The creator of the object can deny access to an unauthorized user.

Object flags are covered as part of the thread discussion here, but this information also pertains to other kernel objects that are obtained by using one of the Windows **Create** functions.

Until now, threads have been created in these solutions with a NULL security attribute. This indicates that the thread should be created using the default security and that the returned handle should be inheritable. If you want to change the behavior of the previous example to prevent the thread handle from being inherited or closed, you can use the **SetHandleInformation** function as follows:

```
#define HANDLE_FLAG_INHERIT 0x00000001
#define HANDLE_FLAG_PROTECT_FROM_CLOSE 0x00000002

SetHandleInformation(hThread, HANDLE_FLAG_INHERIT,
HANDLE_FLAG_INHERIT);
SetHandleInformation(hThread, HANDLE_FLAG_PROTECT_FROM_CLOSE,
HANDLE_FLAG_PROTECT_FROM_CLOSE);
```

To change both flags in a single call, you should join the flags by using a bitwise OR operator. After this call, attempting to close the handle by using the **CloseHandle** function results in an exception being raised.

Thread Scheduling and Prioritizing

This section looks at how you can change the scheduling priority of a thread in UNIX and Windows.

Ideally, you want to map Windows priority classes to UNIX scheduling policies and Windows thread priority levels to UNIX priority levels, but unfortunately, it is not this simple. The priority level of a Windows thread is determined by both the priority class of its process and its priority level. The priority class and priority level are combined to form the base priority of each thread.

Every thread in Windows has a base priority level determined by the priority value of the thread and the priority class of its owning process. The operating system uses the base priority level of all executable threads to determine which thread gets the next slice of CPU time. Threads are scheduled in a round-robin fashion at each priority level, and scheduling of threads at a lower level will only take place when there are no executable threads at a higher level.

UNIX offers both round-robin and FIFO scheduling algorithms, whereas Windows uses only round-robin. This does not mean that Windows is less flexible; it just means that any fine-tuning that was performed on thread scheduling in UNIX has to be implemented differently when using Windows. Table 3.8 lists the base priority levels for combinations of priority class and priority value for Windows.

Table 3.8. Process and Thread Priority for Windows

#	Process Priority Class	Thread Priority Level
1	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
2	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
3	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
4	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
4	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
5	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
5	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
5	Background NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
6	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
6	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
6	Background NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
7	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
7	Background NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
7	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
8	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST

#	Process Priority Class	Thread Priority Level
8	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
8	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
8	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
9	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
9	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
9	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
10	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
10	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
11	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
11	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
11	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
12	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
12	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
13	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
14	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
15	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
15	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
16	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
17	REALTIME_PRIORITY_CLASS	-7
18	REALTIME_PRIORITY_CLASS	-6
19	REALTIME_PRIORITY_CLASS	-5
20	REALTIME_PRIORITY_CLASS	-4
21	REALTIME_PRIORITY_CLASS	-3
22	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
23	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
24	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
25	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
26	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
27	REALTIME_PRIORITY_CLASS	3
28	REALTIME_PRIORITY_CLASS	4

#	Process Priority Class	Thread Priority Level
29	REALTIME_PRIORITY_CLASS	5
30	REALTIME_PRIORITY_CLASS	6
31	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL

Managing Thread Priorities in Windows

The Windows API provides a number of functions for managing thread priorities, including the following:

- **GetThreadContext**. This function returns the execution context of the specified thread. The following is an example showing the thread context:

  ```
  CONTEXT context;
  TCHAR szBuffer[128];
  Context.ContextFlags = CONTEXT_FULL | CONTEXT_DEBUG_REGISTERS;
  GetThreadContext( GetCurrentThread(), &context);
  printf("CS=%X, EIP=%X, FLAGS=%X, DR1=%X\n",
  context.SegCs, context.Eip, context.EFlags, context.Dr1);
  ```

- **GetThreadPriority**. This function returns the assigned thread priority level for the specified thread. To see how thread priority affects the system, a simple test, such as the one that follows, could be added to a simple Windows application:

  ```
  SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_LOWEST);
  DWORD dwTicks = GetTickCount();
  for(long i = 0; i < 200000; i ++)
  for(long j = 0; j < 2000; j ++)
  printf("Test time=%ld\n", GetTickCount() – dwTicks);
  ```

  Adjusting the thread priority should yield different time deltas.

- **GetThreadPriorityBoost**. This function retrieves the priority boost control state of the specified thread. Threads have dynamic priority, which is the priority that the scheduler uses to identify which thread will execute. Initially, the priority of a thread is the same as its base priority, but the system may increase or decrease the priority to maintain thread responsiveness. Only threads with a priority between 0 and 15 are eligible for dynamic priority boost.

  The system boosts the dynamic priority of a thread to enhance its responsiveness as follows:

  - When a process that uses NORMAL_PRIORITY_CLASS is brought to the foreground, the scheduler boosts the priority class of the process associated with the foreground window so that it is equal to or greater than the priority class of any background processes. The priority class returns to its original setting when the process is no longer in the foreground. In Microsoft Windows, the user can control the boosting of processes that use NORMAL_PRIORITY_CLASS through Control Panel.
  - When a window receives input, such as timer messages, mouse messages, or keyboard input, the scheduler boosts the priority of the thread that owns the window.
  - When the wait conditions for a blocked thread are satisfied, the scheduler boosts the priority of the thread. For example, when a wait operation associated with disk or keyboard I/O finishes, the thread receives a priority boost.

- **SetThreadIdealProcessor.** This function specifies the preferred processor for a specific thread. The system schedules threads on the preferred processor when possible.
- **SetThreadPriority**. This function changes the priority level for a thread. For details on the different priority levels, see the Windows API reference.
- **SetThreadPriorityBoost**. This function enables or disables dynamic priority boosts by the system.

Example of Converting UNIX Thread Scheduling into Windows

In this example, the thread priority level is set to the lowest level within the given policy or class for UNIX and Windows respectively. For UNIX, lowering the thread priority level requires creating an attribute object prior to instantiating the thread and then setting the policy of the attribute object. After this activity is complete, the thread is created with the modified attribute. Upon successful instantiation of the thread, the priority level is adjusted to the lowest level within the designated policy and class. In UNIX, this is accomplished by a call to **pthread_attr_setschedparam**. When using the Windows API, it is accomplished by a call to **SetThreadPriority**.

UNIX example: Thread scheduling

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

char message[] = "Hello I'm a Thread";
int thread_finished = 0;

void *thread_function(void *arg)
{
printf("thread_function running. Arg was %s\n", (char *)arg);
sleep(4);
printf("Second thread setting finished flag, and exiting now\n");
thread_finished = 1;
pthread_exit(NULL);
}

int main()
{
int count=0, res, min_priority, max_priority;
struct sched_param scheduling_params;
pthread_t a_thread;
void *thread_result;
pthread_attr_t thread_attr;

res = pthread_attr_init(&thread_attr);
if (res != 0)
{
perror("Attribute creation failed");
exit(EXIT_FAILURE);
}

res = pthread_attr_setschedpolicy(&thread_attr, SCHED_OTHER);
if (res != 0)
{
perror("Setting schedpolicy failed");
exit(EXIT_FAILURE);
}

res = pthread_attr_setdetachstate(&thread_attr,
PTHREAD_CREATE_DETACHED);
if (res != 0)
{
perror("Setting detached attribute failed");
exit(EXIT_FAILURE);
}

res = pthread_create(&a_thread, &thread_attr, thread_function, (void
*)message);
if (res != 0)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}

max_priority = sched_get_priority_max(SCHED_OTHER);
min_priority = sched_get_priority_min(SCHED_OTHER);
scheduling_params.sched_priority = min_priority;

res = pthread_attr_setschedparam(&thread_attr, &scheduling_params);
if (res != 0)
{
perror("Setting schedparam failed");
exit(EXIT_FAILURE);
}

(void)pthread_attr_destroy(&thread_attr);
while(!thread_finished)
{
printf("Waiting for thread to finish (%d)\n", ++count);
sleep(1);
}
```

```
printf("Other thread finished, See Ya!\n");
exit(EXIT_SUCCESS);
}
```

(Source File: U_ThreadSched-UAMV3C3.01.c)

Windows example: Thread scheduling

```
#include <Windows.h>
#include <stdio.h>
#include <stdlib.h>

DWORD WINAPI thread_function(LPVOID arg);

char message[] = "Hello I'm a Thread";
int thread_finished = 0;

void main()
{
int count=0;
HANDLE a_thread;
// Create a new thread.
a_thread = CreateThread(NULL, 0, thread_function, (LPVOID)message, 0,
NULL);

if (a_thread == NULL)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}

if (!SetThreadPriority(a_thread, THREAD_PRIORITY_LOWEST))
{
perror("Setting sched priority failed");
exit(EXIT_FAILURE);
}
CloseHandle(a_thread);

while(!thread_finished)
{
printf("Waiting for thread to finished (%d)\n", ++count);
Sleep(1000);
}
printf("Other thread finished, bye!\n");
exit(EXIT_SUCCESS);
}

DWORD WINAPI thread_function(LPVOID arg)
{
printf("thread_function running. Arg was %s\n", (char *)arg);
Sleep(4000);
printf("Second thread setting finished flag, and exiting now\n");
thread_finished = 1;
return 100;
}
```

(Source File: W_ThreadSched-UAMV3C3.01.c)

In the preceding Windows example, the priority level of the thread is adjusted to the lowest level within the priority class of the owning process. If you want to change the priority class as well as the priority level, insert the following code just before the **SetThreadPriority** call:

```
SetPriorityClass(GetCurrentProcess(), PriorityClass)
```

Where, **PriorityClass** will be one of the values shown in Table 3.9.

Table 3.9 summarizes how to change the scheduling priority for a thread and priority class for the owning process.

Table 3.9. PriorityClass Values

PriorityClass	Meaning
ABOVE_NORMAL_PRIORITY_CLASS	Windows Server™ 2003 and Windows® XP: Indicates a process that has priority above NORMAL_PRIORITY_CLASS but below HIGH_PRIORITY_CLASS.
BELOW_NORMAL_PRIORITY_CLASS	Windows Server 2003 and Windows XP: Indicates a process that has priority above IDLE_PRIORITY_CLASS but below NORMAL_PRIORITY_CLASS.
HIGH_PRIORITY_CLASS	Specify this class for a process that performs time-critical tasks that must be executed immediately. The threads of the process preempt the threads of normal or idle priority-class processes. An example is the Task List, which must respond quickly when called by the user, regardless of the load on the operating system. Use extreme care when using the high-priority class because a high-priority class application can use nearly all available CPU time.
IDLE_PRIORITY_CLASS	Specify this class for a process whose threads run only when the system is idle. The threads of the process are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle-priority class is inherited by child processes.
NORMAL_PRIORITY_CLASS	Specify this class for a process with no special scheduling needs.

PriorityClass	Meaning
REALTIME_PRIORITY_CLASS	Specify this class for a process that has the highest possible priority. The threads of the process preempt the threads of all other processes, including the operating system processes, which may be performing important tasks. For example, a real-time process that executes for more than a very brief interval can prevent disk caches from flushing or can cause the mouse to be unresponsive.

Managing Multiple Threads

In the next two examples, numerous threads are created that terminate at random times. Their termination and display messages are then caught to indicate their termination status. Although these examples are contrived, they do illustrate one key point—the semantics of creating multiple threads and waiting for their completion are similar on both platforms.

UNIX example: Multiple threads

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 5

void *thread_function(void *arg)
{
int t_number = *(int *)arg;
int rand_delay;
printf("thread_function running. Arg was %d\n", t_number);
// Seed the random-number generator with current time so that
// the numbers will be different each time function is run.
srand( (unsigned)time(NULL));
// random time delay from 1 to 10
rand_delay = 1+ 9.0*(float)rand()/(float)RAND_MAX;
sleep(rand_delay);
printf("See Ya from thread #%d\n", t_number);
pthread_exit(NULL);
}

int main()
{
int res;
pthread_t a_thread[NUM_THREADS];
void *thread_result;
int multiple_threads;
for(multiple_threads = 0; multiple_threads < NUM_THREADS;
multiple_threads++)
{
res = pthread_create(&(a_thread[multiple_threads]), NULL,
thread_function,(void *)&multiple_threads);

if (res != 0)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}
sleep(1);
}

printf("Waiting for threads to finish…\n");

for(multiple_threads = NUM_THREADS - 1; multiple_threads >= 0;
multiple_threads--)
{
res = pthread_join(a_thread[multiple_threads], &thread_result);
if (res == 0)
{
printf("Another thread\n");
}
else
{
perror("pthread_join failed");
}
}

printf("All done\n");
exit(EXIT_SUCCESS);
}
```

(Source File: U_MultiThread-UAMV3C3.01.c)

Windows example: Multiple threads

```
#include <Windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NUM_THREADS 5

DWORD WINAPI thread_function(LPVOID arg) {
int t_number = *(int *)arg;
int rand_delay;
printf("thread_function running. Arg was %d\n", t_number);

// Seed the random-number generator with current time so that
// the numbers will be different each time function is run.
srand((unsigned)time(NULL));

// random time delay from 1 to 10
rand_delay = 1 + (rand() % 10);
Sleep(rand_delay*1000);
printf("See Ya from thread #%d\n", t_number);
return 100;
}

void main()
{
HANDLE a_thread[NUM_THREADS];
int multiple_threads;

for(multiple_threads = 0; multiple_threads < NUM_THREADS;
multiple_threads++)
{
// Create a new thread.
a_thread[multiple_threads] =
CreateThread(NULL, 0, thread_function,(LPVOID)&multiple_threads,
0,NULL);
if (a_thread[multiple_threads] == NULL)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}
Sleep(1000);
}
printf("Waiting for threads to finish...\n");

for(multiple_threads = NUM_THREADS - 1; multiple_threads >= 0;
multiple_threads--)
{
if (WaitForSingleObject(a_thread[multiple_threads], INFINITE) ==
WAIT_OBJECT_0)
{
printf("Another thread\n");
}
else
{
perror("WaitForSingleObject failed");
}
}

printf("All done\n");
exit(EXIT_SUCCESS);
}
```

(Source File: W_MultiThread-UAMV3C3.01.c)

I/O Completion Ports

There should always be enough live threads to fully use the available CPUs, but there should never be so many threads that the overhead becomes too large. Multiplexing a large number of clients across a smaller number of live threads is difficult for an application to do. The application cannot always know when a given thread is going to block; and without this knowledge, it cannot activate another thread to take its place. To solve this problem and make it easy for programmers to write efficient and scalable applications, Windows provides a mechanism called the I/O completion port.

An I/O completion port is designed for use with overlapped I/O. A completion port is created with the **CreateIoCompletionPort** function.

```
HANDLE CreateIoCompletionPort(
HANDLE FileHandle,
HANDLE ExistingCompletionPort,
DWORD CompletionKey,
DWORD NumberOfConcurrentThreads
);
```

The **CreateIoCompletionPort** function associates the port with multiple file handles. When asynchronous I/O initiated on any of these file handles completes, an I/O completion packet is queued to the port. This combines the synchronization point for multiple file handles into a single object. If each file handle represents a connection to a client (usually through a named pipe or socket), a handful of threads can manage I/O for any number of clients by waiting on the I/O completion port. Instead of directly waiting for overlapped I/O to complete, these threads use **GetQueuedCompletionStatus** to wait on the I/O completion port. Any thread that waits on a completion port becomes associated with that port. The Windows kernel keeps track of the threads associated with an I/O completion port.

The **WaitForMultipleObjects** function can produce similar behavior, but the most important property of I/O completion ports is the controllable concurrency they provide. The concurrency value of an I/O completion port is specified when it is created. This value limits the number of runnable threads associated with the port; after the number of runnable threads exceeds the concurrency value, the rest of the threads are blocked. As a result, when a thread calls the **GetQueuedCompletionStatus** function, it only returns when a completed I/O is available and the number of runnable threads associated with the completion port is less than the concurrency of the port. Because there is one central synchronization point for all the I/O, a small pool of worker threads can service many clients.

Unlike the other Windows synchronization objects, threads that block on an I/O completion port, by using **GetQueuedCompletionStatus**(), unblock in last-in-first-out (LIFO) order.

A dozen threads can easily service a large set of clients by thread pooling, although this will vary depending on how often each transaction needs to wait.

Note For more information on writing multithreaded scalable applications, refer to MSDN at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndllpro/html/msdn_scalabil.asp.

Chapter 4: Developing Phase: Memory and File Management

This chapter discusses the similarities and differences in memory and file management between the Microsoft® Windows® application programming interface (API) and UNIX. It also provides various memory and file management-related functions and APIs that are available in both environments.

Memory Management

For developers, memory management has always been one of the most important and interesting aspects of any operating system. Like UNIX, the Microsoft Windows operating system provides the standard heap management functions, as well as functions to manage memory on a thread basis. The basic functional mapping is covered in the next sections.

The information provided in this chapter will help you understand the memory and file management implementations in the UNIX and Windows applications. You will also be able to identify the areas that may need to be modified for compatibility in the Windows environment.

Heap

The *heap* is an area of memory that is separate from the program code and the stack. It is reserved for the memory allocation needs of the program that Windows provides similar to UNIX with respect to heap management. The standard C runtime in Windows includes comparable functions for such UNIX functions as **calloc**, **malloc**, **realloc**, and **free**. Windows has additional functions that may not be available in UNIX, which are covered briefly in the following sections.

Thread Local Storage

Thread Local Storage (TLS) is another area of memory management that defines memory on a per-thread basis. All threads of a process share its virtual address space, where the static and global variables are shared by all threads in the process, and the local variables of a function are unique to each thread that runs the function. With TLS*,* you can provide unique data for each thread that the process accesses by using a global index. One thread allocates the index, which can be used by the other threads to retrieve the unique data associated with the index.

The typical scenario in which TLS is used in Windows is within a dynamic-linked library (DLL), but this is not its only possible use. In the case of the DLL scenario, the use of TLS includes the following details:

- When a DLL attaches to a process, the DLL uses **TlsAlloc** to allocate a TLS index. The DLL then allocates some dynamic storage to be used exclusively by the initial thread of the process. It uses the TLS index in a call to the **TlsSetValue** function to store the address in the TLS slot. This concludes the per-thread initialization for the initial thread of the process. The TLS index is stored in a global or static variable of the DLL.
- Each time the DLL attaches to a new thread of the process, the DLL allocates some dynamic storage for the new thread and uses the TLS index in a call to **TlsSetValue** to store the address in the TLS slot. This concludes the per-thread initialization for the new thread.
- Each time an initialized thread makes a DLL call requiring the data in its dynamic storage, the DLL uses the TLS index in a call to **TlsGetValue** to retrieve the address of the dynamic storage for that thread.

The following functions are used to manage TLS in UNIX and Windows:

- **Allocating memory**. In the Windows environment, the **TlsAlloc** function allocates a TLS index. A TLS index is used by a thread to store and retrieve values that are local to the thread. The minimum number of indexes available to each process is defined by TLS_MINIMUM_AVAILABLE. TLS indexes are not valid across process boundaries. The prototype of the function is as follows:

  ```
  DWORD TlsAlloc(void);
  ```

  In the UNIX environment, **pthread_key_create** creates a thread-specific data key visible to all the threads in the process. Upon key creation, the value NULL is associated with the new key in all active threads. An optional destructor function may be associated with each key value. The prototype of the function is as follows:

  ```
  int pthread_key_create(pthread_key_t *key, void (*destructor, void*));
  ```

- **Deleting memory**. In the Windows environment, **TlsFree** releases a TLS index. This, however, does not release the data allocated and set in the TLS index slot. The prototype of the function is as follows:

  ```
  BOOL TlsFree(DWORD dwTlsIndex);
  ```

  In the UNIX environment, the **pthread_key_delete** function deletes the thread-specific data key.

  ```
  int pthread_key_delete(pthread_key_t key);
  ```

- **Storing a value**. In the Windows environment, the **TlsSetValue** function stores memory in a TLS index. The prototype of the function is as follows:

  ```
  BOOL TlsSetValue(DWORD dwTlsIndex, LPVOID lpTlsValue);
  ```

  In the UNIX environment, the **pthread_setspecific** function associates a thread-specific value with the key. The prototype of the function is as follows:

  ```
  int  pthread_setspecific(pthread_key_t key, const void *value);
  ```

- **Retrieving a value**. In the Windows environment, the **TlsGetValue** function returns a memory element stored in a specified TLS index. The prototype of the function is as follows:

  ```
  LPVOID TlsGetValue(DWORD dwTlsIndex);
  ```

  In the UNIX environment, the **pthread_getspecific** function returns the values currently bound to the specific key. The prototype of the function is as follows:

  ```
  void *pthread_getspecific(pthread_key_t key);
  ```

Note Additional information is available at http://msdn.microsoft.com/library/en-us/winprog/winprog/windows_api_reference.asp.

Thread Local Storage (TLS) Example

The following section shows a portion of an example application. It illustrates allocation and access to a memory space on a per-thread basis. First, the main thread of the process allocates a memory slot. The memory slot is then accessed and modified by a child thread. If several instances of the thread are active, each thread procedure will have a unique TLS index value to ensure the separation and isolation of data and state.

UNIX example: Using TLS with pthread library

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
int main(void)
{
      pthread_key_t k1;
      int ret;
      int val = 100;
      int *pval = NULL;
      ret = pthread_key_create(&k1, NULL);
      if (ret)
      {
      printf("pthread_key_create: %s\n", strerror(ret));
      return -1;
      }
      ret = pthread_setspecific(k1, (void *) &val);
      if (ret)
      {
         printf("pthread_setspecific: %s\n", strerror(ret));
         return -2;
      }
      pval =   (int*)pthread_getspecific(k1);
      if(pval == NULL)
      {
         printf("pthred_getspecific: NULL returned");
         return -3;
      }
      printf("pthread_getspecific value: %d\n",*pval);
      ret = pthread_key_delete(k1);
      if (ret)
    {
       printf("pthread_key_delete: %s\n", strerror(ret));
       return -4;
      }
      return 0;
}
```

Windows example: Using TLS with Platform SDK

```
DWORD TLSIndex = 0;
DWORD WINAPI ThreadProc( LPVOID lpData)
{
HWND hWnd = (HWND) lpData;
LPVOID lpVoid = HeapAlloc( GetProcessHeap(), 0, 128 );
TlsSetValue( TLSIndex, lpVoid );
// Do your processing on the memory within the thread here…
HeapFree( GetProcessHeap(), 0, lpVoid );
Return(0);
}

LRESULT CALLBACK WndProc( HWND …
{
switch( uMsg )
{
case WM_CREATE:
TLSIndex = TlsAlloc();
// Start your threads using CreateThread…
Break;
Case WM_DESTROY:
TlsFree( TLSIndex );
Break;
Case WM_COMMAND:
Switch( LWORD( wParam ))
{
case IDM_TEST:
// Do something with the TLS value by a call to TlsGetValue(DWORD)
break;
}
}
}
```

Memory-Mapped Files

Memory-mapped files offer a unique memory management feature that allows applications to access files on disk in the same way they access dynamic memory. It associates a file's contents with a portion of the virtual address space of a process. It can be used to share a file or memory between two or more processes. Windows supports memory-mapped files and memory-mapped page files. Memory-mapped page files are covered in the “Shared Memory” section as part of an exercise to port System V interprocess communication (IPC) shared memory to Windows using memory-mapped files. Creating and using shared memory in UNIX and Windows are conceptually the same activities, but syntactically different.

UNIX example: Creating and mapping a shared memory area

```
if ( (fd = open("/dev/zero", O_RDWR)) < 0)
err_sys("open error");
if ( (area = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0))
== (caddr_t) -1)
err_sys("mmap error");
close(fd); // can close /dev/zero now that it's mapped
```

Windows example: Creating and mapping a shared memory area

```
hMapObject = CreateFileMapping(
INVALID_HANDLE_VALUE, // use paging file
NULL, // no security attributes
PAGE_READWRITE, // read/write access
0, // size: high 32-bits
SHMEMSIZE, // size: low 32-bits
"dllmemfilemap"); // name of map object

if (hMapObject != NULL)
{
// Get a pointer to the file-mapped shared memory.
lpvMem = MapViewOfFile(
hMapObject, // object to map view of
FILE_MAP_WRITE, // read/write access
0, // high offset: map from
0, // low offset: beginning
0); // default: map entire file
if (lpvMem == NULL)
{
CloseHandle(hMapObject);
}
}
```

Note For details on the **CreateFileMapping** and **MapViewOfFile** functions, refer to the Windows API documentation.

Shared Memory

Shared memory permits two or more threads or processes to share a region of memory. Interprocess communication (IPC) through shared memory provides the best performance among all the IPC mechanisms. This is because instead of copying the data, the same physical area of memory is accessed by both the client and the server. Windows does not support the System V IPC mechanisms for shared memory (the **shm*** APIs). However, it does support memory-mapped files and memory-mapped page files, which you can use as an alternative to the **shm*** APIs.

Note System V IPC support is available with MKS and Cygwin on Win32®/Win64.

The **GlobalAlloc** functions can also be used on Windows for sharing memory. The **GlobalAlloc** function allocates the specified number of bytes from the heap, which can be shared among processes.

Note Additional information on **GlobalAlloc** is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/globalalloc.asp.

Synchronizing Access to Shared Resources

The technical challenge of using shared memory is to ensure that the server and client do not attempt to access the shared resource simultaneously. This is particularly troublesome if one or both are writing to the same shared memory area. For example, if the server is writing to the shared memory, the client must not try to access the data until the server has completed the write operation.

To address this, several forms of synchronization are available in Windows. These are:

- Semaphore
- Mutex
- Event
- Critical section

UNIX has two of these mechanisms—the semaphore and the mutex—as well as an additional mechanism called file locking.

Note These mechanisms are also discussed in Chapter 3, “Developing Phase: Process and Thread Management” of this volume.

The first three mechanisms have two states: signaled and nonsignaled. The synchronization object is considered busy when it is in a nonsignaled state. When the object is busy, a waiting thread will block until the object switches to a signaled state. At this time, the pending thread continues to execute. The last form of synchronization is the critical section object. The critical section object is only for synchronizing threads within a single process. This synchronization mechanism only works for a single instance of the example application. Nonetheless, you can still consider its use as an IPC synchronization mechanism. This form of synchronization is appropriate for cases where you want to migrate your existing application from a multiprocessor architecture to a single multithreaded processor architecture.

The synchronization objects should be used carefully to prevent deadlocks. Deadlocks occur when some subset of the threads is waiting on one another so that none can execute.

Note Applications that need to control the size of the stack or heap space can use linker switches for this purpose. The default size for both the stack and heap on Windows is 1 MB. Use the /STACK linker option to set the size of the stack and the /HEAP linker option to set the heap size. Both options take the size in bytes. Additional information on how to use these linker options is available at

http://msdn2.microsoft.com/en-us/library/f90ybzkh.aspx or

http://msdn2.microsoft.com/en-us/library/8cxs58a6(en-US,VS.80).aspx.

Further Reading on Memory Management

For further information about memory management, refer to the following:

- Richter, Jeffrey. *Programming Applications for Microsoft Windows*, *Fourth Edition*. Redmond, WA: Microsoft Press®, 1999. (See Part III, Chapters 13–18.)
- Stevens, W. Richard. *Advanced Programming in the UNIX Environment*. Reading, MA: Addison-Wesley Publishing Co., 1992.

File Management

This section discusses the differences in file handling, file access, file control, and directory operations. You can use this information to understand the file management implementations in UNIX and Windows applications. You can also identify the areas that may need to be modified for compatibility within the Windows environment.

Every program that runs from the UNIX shell opens three standard files. These files have integer file descriptors, provide the primary means of communication between the programs, and exist for as long as the process runs. You associate other file descriptors with files and devices using the **open** system call. Table 4.1 lists the UNIX standard file descriptors.

Table 4.1. UNIX Standard File Descriptors

File	File Descriptor	Description
Standard input	0	Standard input file provides a way to send data to a process. By default, the standard input is read from the keyboard.
Standard output	1	Standard output file provides a means for the program to output data. By default, the standard output goes to the display.
Standard error	2	Standard error is where the program reports any errors that occurred during the program execution. By default, the standard error goes to the display.

In Windows, when a program begins execution, the startup code automatically opens the following files (streams):

- Standard input (pointed to by **stdin**)
- Standard output (pointed to by **stdou**t)
- Standard error (pointed to by **stderr**)

These stream files are directed to the console (keyboard and screen) by default. Use **freopen** to redirect **stdin**, **stdout**, or **stderr** to a disk file or a device.

The **stdout** and **stderr** stream files are flushed whenever they are full or, if you are writing to a character device, after each library call. If a program terminates abnormally, output buffers may not be flushed, resulting in loss of data. Use **fflush** or **_flushall** to ensure that the buffer associated with a specified file or that all open buffers are flushed to the operating system, which can cache data before writing it to disk. The commit-to-disk feature ensures that the flushed buffer content is not lost in the event of a system failure.

Low-Level File Access

The low-level I/O functions directly invoke the operating system for lower-level operation, instead of the operation provided by standard (or stream) I/O. Function calls relating to low-level input and output do not buffer or format data. Low-level I/O functions can access the standard streams opened at program startup using the standard file descriptors. They deal with bytes of information, which implies that you are using binary files, not text files. Instead of file pointers, you use low-level file handles or file descriptors, which give a unique integer number to identify each file.

The read and write method used in UNIX have equivalents in the Windows environment as _read and _write. These methods can be used to read data from standard input and write data to standard output.

Information about the prototype and additional details about the low-level I/O routines available in Windows can be found at

http://msdn2.microsoft.com/en-us/library/40bbyw78.aspx.

Note Just as in UNIX, the Windows **cmd.exe** shell redirects standard error from the command line with the **2>** operator.

Standard (Stream) File Access

The standard or stream I/O functions process data in different sizes and formats, and from single characters to large data structures. They also provide buffering, which can improve performance. These routines affect only buffers created by the run-time library routines and have no effect on buffers created by the operating system. The standard I/O library and its header file stdio.h provide low-level file I/O system calls. This library is part of ANSI standard C, so they can be ported directly to Windows.

Both UNIX and Windows support the standard I/O library functions for several file access purposes.

Note Additional information on the list of I/O library functions and their prototypes is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore98/HTML/_crt_stream_i.2f.o.asp.

ioctl Calls

The **ioctl** function performs a variety of control operations on devices and streams. For nonstream files, the operations performed by this call are device-specific control operations. The prototype of the **ioctl** function is as follows.

```
int ioctl(int fildes, int request, /* arg */ …);
```

In Windows, a subset of the operations on a socket is provided by the **ioctlsocket** function. The **ioctlsocket** function supports only the SIOCATMARK command and does not have a command parameter equivalent to the FIOASYNC parameter of the **ioctl** function.

Windows ioctlsocket

The **ioctlsocket** function controls the I/O mode of a socket.

```
int ioctlsocket(SOCKET s, long cmd, u_long FAR *argp);
```

The **ioctlsocket** function can be used on any socket in any state. It is used to set or retrieve operating parameters associated with the socket, independent of the protocol and communications subsystem. Table 4.2 lists the supported commands to use in the command parameter and their semantics.

Table 4.2. ioctlsocket Parameters and Semantics

Command	Description

FIONBIO	Use with a nonzero **argp** parameter to enable the nonblocking mode of sockets. Set the **argp** parameter to zero if nonblocking is to be disabled. The **argp** parameter points to an unsigned long value. When created, a socket operates in blocking mode by default (nonblocking mode is disabled). This is consistent with BSD sockets. The **WSAAsyncSelect** and **WSAEventSelect** functions automatically set a socket to nonblocking mode. If **WSAAsyncSelect** or **WSAEventSelect** has been issued on a socket, then any attempt to use **ioctlsocket** to set the socket back to blocking mode will fail with WSAEINVAL. To set the socket back to blocking mode, an application must first disable **WSAAsyncSelect** by calling **WSAAsyncSelect** with the lEvent parameter set to zero or by calling **WSAEventSelect** with the lNetworkEvents parameter set to zero. The prototypes of the **WSAAsyncSelect** and **WSAEventSelect** functions are as follows**:**
	`int WSAAsyncSelect(SOCKET s, HWND hWnd, unsigned int wMsg, long lEvent)`
	`int WSAEventSelect(SOCKET s, WSAEVENT hEventObject, long lNetworkEvents)`

FIONREAD	Use to determine the amount of data pending in the input buffer of the network that can be read from sockets. The **argp** parameter points to an unsigned long value in which **ioctlsocket** stores the result. FIONREAD returns the amount of data that can be read in a single call to the **recv** function, which may not be the same as the total amount of data queued on the socket. If s is message oriented (for example, type SOCK_DGRAM), FIONREAD still returns the amount of pending data in the network buffer. However, the amount that can actually be read in a single call to the **recv** function is limited to the data size written in the **send** or **sendto** function call. The prototypes of the **recv**, **send**, and **sendto** functions are as follows:
	`int recv (int s, void *buf, int len, int flags)`
	`int send (SOCKET s, const char FAR* buf, int len, int flags)`
	`int sendto (SOCKET s, const char FAR* buf, int len, int flags, const struct SOCK_ADDR* to, int tolen)`

SIOCATMARK	Use to determine whether all out-of-band (OOB) data has been read. For a discussion of OOB data, refer to the "Windows Sockets 1.1 Blocking Routines and EINPROGRESS" section in the Microsoft Platform SDK: Windows Sockets 2 reference available on MSDN®. This applies only to a stream-oriented socket (for example, type SOCK_STREAM) that has been configured for inline reception of any OOB data (SO_OOBINLINE). If no OOB data is waiting to be read, the operation returns TRUE. Otherwise, it returns FALSE, and the next **recv** or **recvfrom** performed on the socket will retrieve some or all of the data preceding the mark. The application should use the SIOCATMARK operation to determine whether any data remains. If there is any typical data preceding the urgent (out-of-band) data, it will be received in order. A **recv** or **recvfrom** never mixes OOB and typical data in the same call. The **argp** parameter points to an unsigned long value in which **ioctlsocket** stores the boolean result.

File Control

File control in UNIX is implemented using the **fcntl** function. The **fcntl** function performs one of a number of miscellaneous operations on file descriptors.

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

Table 4.3 lists the commands and semantics.

Table 4.3. File Control Commands and Semantics

Command	**Description**
F_DUPFD	Find the lowest numbered file descriptor available that is equal to or greater than **arg** and make it a copy of **fd**. On success, the new descriptor is returned.
F_GETFD	Read the close-on-exec flag. On success, the value of the flag is returned.
F_SETFD	Set the close-on-exec flag to the value specified by the FD_CLOEXEC bit of **arg**.
F_GETFL	Read the flags of the descriptor. All flags, as set by **open**, are returned.
F_SETFL	Set the flags of the descriptor to the value specified by **arg**. O_APPEND, O_NONBLOCK, and O_ASYNC may be set; the other flags are unaffected.
F_GETLK, F_SETLK, and F_SETLKW	These commands are used to manage discretionary file locks. The third argument **lock** is a pointer to a struct flock, which may be overwritten by this call.
F_GETLK	Return the first flock structure that prevents the requested lock from being created or set the l_type field of the lock to F_UNLCK if there is no obstruction.
F_SETLK	The lock is set (when l_type is F_RDLCK or F_WRLCK) or cleared (when it is F_UNLCK). If the lock is held by another user, this call returns -1 and sets **errno** to EACCES or EAGAIN.
F_SETLKW	Like F_SETLK, but instead of returning an error, wait for the lock to be released. If a signal that is to be caught is received while **fcntl** is waiting, it is interrupted and (after the signal handler has returned) returns immediately with a return value of -1 and **errno** set to EINTR.
F_GETOWN, F_SETOWN, F_GETSIG and F_SETSIG	These commands are used to manage I/O availability signals.
F_GETOWN	Get the process ID or process group currently receiving SIGIO and SIGURG signals for events on file descriptor **fd**. Process groups are returned as negative values.
F_SETOWN	Set the process ID or process group that will receive SIGIO and SIGURG signals for events on file descriptor **fd**. Process groups are specified using negative values. F_SETSIG can be used to specify a different signal instead of SIGIO.

Command	Description
F_GETSIG	Get the signal sent when input or output becomes possible. A value of zero means SIGIO is sent. Any other value (including SIGIO) is the signal sent instead. In this case, additional information is available to the signal handler if installed with SA_SIGINFO.
F_SETSIG	Sets the signal to be sent when input or output becomes possible. A value of zero sends the default SIGIO signal. Any other value (including SIGIO) is the signal to send instead. In this case, additional information is available to the signal handler if installed with SASIGINFO.

In Windows, the following equivalent functions are available for some, but not all, of the UNIX **fcntl** commands:

- Use the **_dup** function in Windows for the F_DUPFD command in **fcnt**l function in UNIX.
- Use the **LockFile**, **LockFileEx**, and **UnLockFile** functions in Windows for the F_SETLK and F_SETLKW commands of **fcntl** function in UNIX. The following example demonstrates this.

UNIX example: Using fcntl

The following sample opens a file, sets the read lock, and unlocks the file. If any errors occur, an error message is output to the standard error file descriptor.

```
#include <unistd.h>
#include <fcntl.h>

int main()
{
struct flock l;
int fd = open("/tmp/locktest", O_RDWR|O_CREAT, 0644);

if (fd < 0)
{
perror("file open error");
exit(1);
}

l.l_type = F_RDLCK;
l.l_whence = SEEK_SET;
l.l_start = 0;
l.l_len = 0;
if (fcntl(fd, F_SETLK, &l) == -1)
{
perror("fcntl error – F_RDLCK");
exit(1);
}

l.l_type = F_UNLCK;
if (fcntl(fd, F_SETLK, &l) == -1)
{
perror("fcntl error – F_UNLCK");
exit(1);
}

exit(0);
}
```

(Source File: U_Fcntl-UAMV3C4.02.c)

Windows example: Using fcntl

The following sample opens a file, sets the read lock, and unlocks the file. If any errors occur, an error message is output to the console.

```
#include <stdio.h>
#include <fcntl.h>
#include <windows.h>
#include <io.h>

int main()
{
HANDLE hFile;
DWORD dwBytesRead, dwPos;
BOOL fResult;

// Open the existing file.
hFile = CreateFile("ONE.TXT", // open ONE.TXT
GENERIC_READ, // open for reading
0, // do not share
NULL, // no security
OPEN_EXISTING, // existing file only
FILE_ATTRIBUTE_NORMAL, // normal file
NULL); // no attr. template

if (hFile == INVALID_HANDLE_VALUE)
{
printf("Could not open ONE.TXT\n"); // process error
}

// Lock the file
fResult = LockFile(hFile, dwPos, 0, dwPos + dwBytesRead, 0);
if (fResult == 0)
{
printf("Could not lock ONE.TXT\n");
}
// Unlock the file
UnlockFile(hFile, dwPos, 0, dwPos + dwBytesRead, 0);
if (fResult == 0)
{
printf("Could not unlock ONE.TXT\n");
}

// Close file.
CloseHandle(hFile);
return(0);
}
```

(Source File: W_Fcntl-UAMV3C4.02.c)

Directory Operation

Directory operations involve calling the appropriate functions to perform directory scanning or to list the contents of a directory. Directory scanning involves traversing a directory hierarchy.

Table 4.4 provides details about the related functions for directory operations in UNIX and their replacements in Windows.

Table 4.4. Directory Operations Functions in UNIX and Their Replacements in Windows

UNIX Functions	Description	Suggested Replacement in Windows
getcwd **getwd**	Gets the current working directory.	The **_getcwd function** does the same action. The prototype of the function is as follows: `char *_getcwd(char *buffer, int maxlen);`
get_current_dir_name	Gets the current working directory.	The **GetCurrentDirectory** function gives the same result. The prototype of the function is as follows: `DWORD GetCurrentDirectory(DWORD nBufferLength, LPTSTR lpBuffer)`

The following examples illustrate additional directory handling functions in UNIX and Windows.

UNIX example: Using directory handling functions

This sample prints out the current directory, and then recurses through subdirectories.

```
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>

void ScanDir(char *dir, int indent)
```

```
{
DIR *dp;
struct dirent *dir_entry;
struct stat stat_info;

if((dp = opendir(dir)) == NULL)
{
fprintf(stderr,"cannot open directory: %s\n", dir);
return;
}
chdir(dir);

while((dir_entry = readdir(dp)) != NULL)
{
lstat(dir_entry->d_name,&stat_info);
if(S_ISDIR(stat_info.st_mode))
{
/* Directory, but ignore . and .. */
if(strcmp(".",dir_entry->d_name) == 0 || strcmp("..",dir_entry->d_name)
== 0)
continue;
printf("%*s%s/\n",indent,"",dir_entry->d_name);
/* Recurse at a new indent level */
ScanDir(dir_entry->d_name,indent+4);
}
else printf("%*s%s\n",indent,"",dir_entry->d_name);
}
chdir("..");

closedir(dp);
}

int main(int argc, char* argv[])
{
char *topdir, defaultdir[2]=".";

if (argc != 2) {
printf("Argument not supplied - using current directory.\n");
topdir=defaultdir;
}
else
topdir=argv[1];
printf("Directory scan of %s\n",topdir);
ScanDir(topdir,0);
printf("done.\n");
exit(0);
}
```

(Source File: U_Dir-UAMV3C4.01.c)

Windows example: Using directory handling functions

This sample prints out the current directory and then recurses through subdirectories. It uses the **FindFirstFile, FindNextFile,** and **FindClose** Windows API functions.

```
#include <windows.h>
#include <stdio.h>

void ScanDir(char *dirname, int indent)
{
BOOL fFinished;
HANDLE hList;
TCHAR szDir[MAX_PATH+1];
TCHAR szSubDir[MAX_PATH+1];
WIN32_FIND_DATA FileData;

// Get the proper directory path
sprintf(szDir, "%s\\*", dirname);

// Get the first file
hList = FindFirstFile(szDir, &FileData);
if (hList == INVALID_HANDLE_VALUE)
{
printf("No files found\n\n");
}
else
{
// Traverse through the directory structure
fFinished = FALSE;
while (!fFinished)
{
// Check the object is a directory or not
if (FileData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
{
if ((strcmp(FileData.cFileName, ".") != 0) &&
(strcmp(FileData.cFileName, "..") != 0))
{
printf("%*s%s\\\n", indent, "", FileData.cFileName);
// Get the full path for sub directory
sprintf(szSubDir, "%s\\%s", dirname, FileData.cFileName);
ScanDir(szSubDir, indent + 4);
}
}
else
printf("%*s%s\n", indent, "", FileData.cFileName);
if (!FindNextFile(hList, &FileData))
{
if (GetLastError() == ERROR_NO_MORE_FILES)
{
fFinished = TRUE;
}
}
}
}
FindClose(hList);
}

void main(int argc, char *argv[])
{
char *pszInputPath;
char pwd[2] = ".";

if (argc < 2)
{
printf("Argument not supplied - using current directory.\n");
pszInputPath = pwd;
}
else
{
pszInputPath = argv[1];
printf("Input Path: %s\n\n", pszInputPath);
}
ScanDir(pszInputPath, 0);

printf("\ndone.\n");
}
```

(Source File: W_Dir-UAMV3C4.01.c)

Raw Device I/O

A raw device can be bound to an existing block device (for example, a disk) and can be used to perform raw I/O with that block device. Raw I/O does not involve the kernel caching that is normally associated with block devices. In UNIX, the system calls for character devices can be used to operate on raw devices. Device-specific files are created by the **mknod** system call. There is an additional system call, **ioctl**, for manipulating the underlying device parameters of special files. The prototype of the **mknod** and **ioctl** functions is as follows:

```
int mknod( char *pathname, mode_t mode, dev_t dev);
int ioctl(int fildes, int request, /* arg */ ...);
```

The following example describes the usage of **mknod** function for creating the device-specific files.

UNIX example: Raw device I/O

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/sysmacros.h>
#include <unistd.h>
int main(int argc,char **argv)
{
            int major =0,minor=0;
            char *path = "test.x";
            int mode = 0666 | S_IFBLK;
            char *end;
            /* get the major and minor numbers for device files */
            major = strtol("01",&end,0);
            if(*end)
            {
                        printf("error in minor number:%d\n",minor);
                        return 1;
            }
            minor = strtol("01",&end,0);
            if(*end)
            {
                        printf("error in minor number:%d\n",minor);
                        return 1;
            }
            /* creating device file */
          if(mknod(path,mode,makedev(major,minor)))
          {
            printf("error in creating the device file:
%s\n",strerror(errno));
            return 2;
            }
            return 0;
}
```

Raw device I/O is common in Windows. Device I/O in Windows can be done with asynchronous I/O (or overlapped I/O). A process can open a file for asynchronous I/O with a call to **CreateFile** by specifying the FILE_FLAG_OVERLAPPED flag in the parameter. When the file is opened for asynchronous I/O, a pointer to an OVERLAPPED structure is passed into the call to **ReadFile** and **WriteFile**. You can also create an event and put the handle in the OVERLAPPED structure; the **wait** functions can then be used to wait for the I/O operation to complete by waiting on the event handle. The prototype of the Createfile, WriteFile, and ReadFile methods are as follows.

```
HANDLE CreateFile(LPCTSTR lpFileName, DWORD dwDesiredAccess, DWORD
dwShareMode, LPSECURITY_ATTRIBUTES lpSecurityAttributes, DWORD
dwCreationDisposition, DWORD dwFlagsAndAttributes, HANDLE
hTemplateFile);
```

```
BOOL WriteFile(HANDLE hFile, LPCVOID lpBuffer, DWORD
nNumberOfBytesToWrite, LPDWORD lpNumberOfBytesWritten, LPOVERLAPPED
lpOverlapped);
```

```
BOOL ReadFile(HANDLE hFile, LPVOID lpBuffer, DWORD
nNumberOfBytesToRead, LPDWORD lpNumberOfBytesRead, LPOVERLAPPED
lpOverlapped);
```

```
BOOL DeviceIoControl(HANDLE hDevice, DWORD dwIoControlCode, LPVOID
lpInBuffer, DWORD nInBufferSize, LPVOID lpOutBuffer, DWORD
nOutBufferSize, LPDWORD lpBytesReturned, LPOVERLAPPED lpOverlapped);
```

Windows example: Raw device I/O

The following example describes the usage of raw device I/O with the Windows API.

```
#include <windows.h>
#include <stdio.h>



int main()
{
 int nDiskNumber = 0;
char buf[64];
memset(buf,0,sizeof(buf));
sprintf ( buf, "\\\\.\\PHYSICALDRIVE%c", (char)nDiskNumber+'0');
//Open the physical disk handle
HANDLE hDiskHandle =   CreateFile ( buf , GENERIC_READ | GENERIC_WRITE
                        , FILE_SHARE_READ | FILE_SHARE_WRITE ,
NULL,OPEN_EXISTING, 0 , NULL );
if (hDiskHandle == INVALID_HANDLE_VALUE)
{
    printf ("PhysicalDisk:: can not open physical disk");
    return 1;
}

int nSide,nTrack,nSector;
nSide = nTrack = nSector = 200 ;

LARGE_INTEGER li;
li.QuadPart = UInt32x32To64 (nSide*nTrack*nSector, 512);
//Move the physical disk pointer to the required position
DWORD dwPos = SetFilePointer ( hDiskHandle, li.LowPart, &li.HighPart,
FILE_BEGIN);
DWORD dwError = GetLastError();
if(dwError != NO_ERROR)
{
      printf("error in moving the file position\n");
      return 1;
}
unsigned long bytesRead;
char inBuffer[512];
memset(inBuffer,0,sizeof(bytesRead));
//Read the data from the physical disk
BOOL result = ReadFile( hDiskHandle,inBuffer,512 , &bytesRead , NULL );
 dwError = GetLastError();
 if( bytesRead <= 0 || dwError)
 {
       printf("error in reading the file\n");
       return 2;
 }
 CloseHandle(hDiskHandle);
 return 0;
}
```

Note More information on raw I/O and file systems in Windows is available at the MSDN Web site.

Chapter 5: Developing Phase: Infrastructure Services

This chapter discusses the potential coding differences between UNIX and Microsoft® Windows® operating systems with respect to infrastructure services.

Infrastructure services are discussed in the following sections:

- Security
- Handles
- Error and Exception Handling
- Handles
- Signals vs. Events
- Interprocess Communication (IPC)
- Networking

This chapter describes the implementation of the preceding infrastructure services in the Windows environment and provides a detailed comparison with the corresponding implementation in UNIX.

Using the information provided in this chapter, you can identify any incompatibilities in your applications in these areas and also learn about the suggested replacement mechanisms for the Windows environment.

Security

The UNIX and Windows security models are quite different. The Windows application programming interface (API) uses the underlying Windows security model. This results in key differences between the way Windows security works and the way UNIX security works. Some of these differences are covered in the “Architectural Differences” section in Chapter 1, “Introduction to Win32/Win64” of this volume. This chapter covers the differences in their respective security models. This section describes various security features available in Windows, such as user-level security and process-level security, and compares them with UNIX. With this information, you can modify the security-related code in your applications to operate in Windows.

User-Level Security

This section discusses the differences in implementing user-level security in UNIX and Windows.

Retrieving the User Name of the Current User

The following examples illustrate the migration of code from UNIX to Windows to retrieve the user name of the current user.

UNIX example: Retrieving the user name of the current user

This example uses the **getlogin** function to retrieve the user name of the user currently logged onto the system.

```
#include <stdio.h>
#include <unistd.h>

main()
{
// Get and display the user name.
printf("User name: %s\n", getlogin());
}
```

(Source File: U_GetUser-UAMV3C5.01.c)

Windows example: Retrieving the user name of the current user

This example uses the **GetUserName** function to retrieve the user name of the current thread. This is the name of the user currently logged on to the system.

The **GetUserNameEx** function can be used to retrieve the user name in a specified format.

```
#include <windows.h>
#include <stdio.h>
#include <lmcons.h>

void main()
{
LPTSTR lpszSystemInfo; // pointer to system information string
DWORD cchBuff = 256; // size of user name
TCHAR tchBuffer[UNLEN + 1]; // buffer for expanded string
lpszSystemInfo = tchBuffer;

// Get and display the user name.
GetUserName(lpszSystemInfo, &cchBuff);
printf("User name: %s\n", lpszSystemInfo);
}
```

(Source File: W_GetUser-UAMV3C5.01.c)

Process-Level Security

In UNIX, access rights are determined by the permissions on files. Each user has a UID, which (unlike Windows) does not have to be unique. A user is logged on to the system when a shell process is running that has the UID of the user. Groups are sets of users. A UNIX group has a Group ID (GID). Every process has a UID and a GID associated with it.

When a user logs on to the system with a user name and a password, UNIX starts a shell with the UID and GID of that user. From then on, all access to files and other resources is controlled by the permissions assigned to the UID and GID or the process. The UIDs and GIDs are configured in two files: /etc/passwd and /etc/group.

Security permissions are applied to files based on users or groups. The permissions that can be granted are read, write, and execute. These permissions are grouped in three sets: the owner of the file, the group of the owner, and everyone else. A full (long) listing for a file shows the file permissions as a group of nine characters that indicate the permissions for owner, group, and everyone. The characters r, w, x, and - are used to indicate read, write, execute, and no permission, respectively. For example, if the owner of a file has all permissions but the group and everyone have only read permission, the string is as follows:

```
rwxr--r--
```

Note Some UNIX implementations have extended the basic security model to include access control lists (ACLs) similar to those used in Windows. However, ACLs are not implemented consistently across all versions of UNIX.

A process can take the identity of another user in order to gain the access permissions of that user or to use resources that may be accessible to the other user. This can be done using the **setuid** function in UNIX. The following example changes the real, effective, and saved-set-UIDs.

UNIX example: Set the UID of the process to a specific user

```
#include <unistd.h>
#include <stdio.h>

int main(void)
{
printf ( "prior to setuid(), uid = %d, effective uid = %d\n"
(int) getuid(), (int) geteuid() );
if ( setuid(25) != 0 )
perror( "setuid() error" );
else
printf ( "after setuid(),     uid = %d, effective uid = %d\n",
(int) getuid(), (int) geteuid() );
    return 0;
}
```

(Source File: U_SetUid-UAMV3C5.01.c)

Windows uses a unified security model that protects all objects from unauthorized access. The system maintains security information for the following:

- **Users.** The people who log on to the system, either interactively by entering a set of credentials (user name and password) or remotely through the network. The security context of every user is represented by a logon session. Each process that the user starts is associated with the logon session of the user.
- **Objects.** The secured resources that a user can access. For example, files, synchronization objects, and named pipes represent kernel objects.

Access Tokens

An access token is a data structure associated with every process that is started by a particular user and is associated with the logon session of that user. The access token identifies who the user is and which security groups he or she belongs to. Although users and groups have human-readable names to ease administration, for performance reasons they are uniquely identified, internally, by security identifiers (SIDs).

Security Descriptors

A security descriptor describes the security attributes of each object. The information in the security descriptor includes the owner of the object, the system access control list (SACL), and a discretionary access control list (DACL). The DACL contains a list of access control entries (ACEs) that defines the access rights for particular users or groups of users. The owner of the object controls the DACL and uses it to determine who should and should not be allowed access to the object, and what rights should be granted to them.

The security descriptor also includes a system access control list (SACL), which is controlled by system administrators. Administrators use SACLs to specify auditing requirements for object access. For example, an administrator can establish a SACL that specifies the generation of an audit log entry whenever a user attempts to delete a particular file.

The sequence of events from the time a user logs on, to the time the user attempts to access a secure object, is as follows:

1. The user logs on with a set of credentials. The system validates these credentials by comparing them against the information maintained in a security database (or Microsoft Active Directory® directory service).
2. If the details of the user are authenticated, the system creates a logon session that represents the security context for the user. Every process created on behalf of the user (starting with the Windows shell process) contains an access token that describes the security context of the user.
3. Every process subsequently started by the user is passed a copy of the access token. If one process results in additional processes, all child processes obtain a copy of the access token and are associated with the single logon session of the user.
4. When a process (acting on behalf of the user) attempts to open a secure object such as a file, the process must initially obtain a handle to the object. For example, when attempting to open a file, the process calls the **CreateFile** function. The process specifies a set of access rights on the call to **CreateFile**.
5. The security system accesses the security descriptor of the object and uses the list of ACEs contained in the DACL to find a group or user SID that matches the one contained in the access token of the process. When this task is complete, the user is either denied access to the object (if a deny ACE is located) or the user is granted a specific set of access rights to the object. The granted rights may be the same as the rights initially requested or they may be a subset of the rights initially requested. For example, the **CreateFile** call can request read and write access to a file, but the DACL may allow only read access.

Impersonation

When a thread within a process attempts to access a secured object, the security context that represents the user who is making the access attempt is normally obtained from the process-level access token. You can, however, associate a temporary access token with a specific thread.

For example, within a server process, you can impersonate the security context of a client. The act of impersonation associates a temporary access token with the current thread. The temporary impersonation access token represents the security context of the client. As a result, the server thread uses the security context of the user when it attempts to access any secured object. When the temporary access token is removed from the thread, impersonation ceases and subsequent resource access reverts to using the process-level access token.

The **ImpersonateLoggedOnUser** function can be used for this. The user is represented by a token handle.

```
BOOL ImpersonateLoggedOnUser(
HANDLE hToken
);
```

Here, hToken is a handle to a primary or impersonation access token that represents a logged-on user. This can be a token handle returned by a call to the **LogonUser**, **CreateRestrictedToken**, **DuplicateToken**, **DuplicateTokenEx**, **OpenProcessToken**, or **OpenThreadToken** functions.

The impersonation lasts until the thread exits or until it calls **RevertToSelf**. The calling thread does not need to have any particular privileges to call **ImpersonateLoggedOnUser**.

Handles

This section discusses the differences in implementing system handles in UNIX versus Windows. This section covers the following topics:

- Socket handles
- File handles
- Output buffer or event queue handling

By understanding the implementation of handles in Windows and UNIX, you will be able to identify the unsupported handle-specific features in the UNIX application and the replacement mechanisms in the Windows environment specific to signal and error handling.

Socket Handles

In UNIX, socket handles are small, non-negative integers. Socket handles can be passed to most of the low-level Portable Operating System Interface (POSIX) input/output (I/O) functions. For example:

```
read(s, buffer, buffer_len);
```

In the earlier example, *s* could either be a socket or a file handle. It is common to use **read** instead of **recv** to read data from a socket, for example. Similarly, the **write** function call is equivalent to **send** and **sendto**.

Windows defines a new unsigned data type SOCKET that may take any value in the range 0 to INVALID_SOCKET–1, where INVALID_SOCKET is a predefined value for a nonexistent socket. Because the SOCKET type is unsigned, compiling existing source code from a UNIX environment may lead to compiler warnings about signed/unsigned data type mismatches.

A socket handle can optionally be a file handle in Windows Sockets 2. It is possible to use socket handles with the **ReadFile**, **WriteFile**, **ReadFileEx**, **WriteFileEx**, **DuplicateHandle**, and other functions. However, for an application to run over the widest possible number of service providers, it should not assume that socket handles are file handles.

File Handles

In UNIX, a file handle is an opaque number that is used to uniquely identify a file or other file system objects.

The fact that the file handle is opaque means that no information can be obtained from the file by inspecting the contents of the file handle. The only operations that can be done with the file handle are to copy and compare it for equality with another file handle.

The traditional contents of a file handle are:

- An identifier for the file system, such as the device number from which the file system is mounted.
- An identifier for the inode within the file, such as the inode number.
- A field to indicate when an inode has been reused; this is typically called a generation number for the inode.

In Windows, the file handle is used to identify a file. When a file is opened by a process using the **CreateFile** function, a file handle is associated with it until either the process terminates or the handle is closed using the **CloseHandle** function.

Each file handle is generally unique to each process that opens a file. The only exceptions to this are when a file handle held by a process is duplicated or when a child process inherits the file handles form the parent process. These file handles are unique, but they refer to a single, shared file object.

Note Although the file handles are typically private to a process, the file data that the file handles point to is not. Therefore, processes and threads that share the same file must synchronize their access. For most operations on a file, a process identifies the file through its private pool of handles.

Output Buffer or Event Queue Handling

The operating system may respond to events immediately or put them in an **EventQueue** for later processing. The **select** function can be used in UNIX for this. This function indicates which of the specified file descriptors are ready for reading or writing or which have an error condition pending. If none of the specified file descriptors is ready for reading/writing or has an error condition pending, the **select** function keeps blocking itself for a predefined timeout interval until one of the file descriptors is ready. The **select** function supports regular files, terminal and pseudo-terminal devices, stream-based files, FIFOs, and pipes.

In Windows, this is possible with the **WaitForMultipleObjects** or **WaitForSingleObject** functions as well. The **WaitForMultipleObjects** function determines whether the given input objects meet the wait criteria. It returns a value when either of the specified objects is in the signaled state or when the time-out interval elapses. **WaitForSingleObject** also behaves in the same way as **WaitForMultipleObjects** except that it works with the single input object. If the criteria are not met, the calling thread enters the wait state. It does not use any processor cycles while waiting for the criteria to be met.

These functions can specify handles of any of the following object types as the input parameter:

- Change notification
- Console input
- Event
- Job
- Memory resource notification
- Mutex
- Process
- Semaphore
- Thread
- Waitable timer

Use caution when calling the **wait** functions and code that directly or indirectly creates windows. If a thread creates any window, it must process messages. Message broadcasts are sent to all windows in the system. A thread that uses a **wait** function with no time-out interval may cause the system to enter a deadlock. Therefore, if you have a thread that creates windows, use the **MsgWaitForMultipleObjects** or **MsgWaitForMultipleObjectsEx** function instead of **WaitForMultipleObjects**. There is no corresponding function to **WaitForSingleObject** in this context.

The **WaitForMultipleObjects** function takes a **HANDLE***, but HANDLEs to files and sockets are not allowed to be in that array. Instead, asynchronous I/Os can take synchronization objects as input parameters, which in turn can be made to wait using the **WaitForMultipleObjects** function. The **WaitForMultipleObjectEx** can wake the thread automatically with a special result when any asynchronous I/O scheduled by the calling thread is completed.

An alternative solution would be that all communication is performed through file descriptors. This requires modifying the operating system for something as simple as adding a new **EventQueue**. **WinSocket** solves that by putting all socket events in the window **EventQueue**. All petitions made to the operating system are responded to through the event queue. Each event has all the necessary information to know what needs to be done with the result.

Error and Exception Handling

In UNIX, when an error occurs in a function, a negative value is often returned and the integer **errno** is usually set to a value that gives information about the error. The file errno.h defines the variable **errno** and the constants for each value that **errno** can assume. The application can retrieve and set the last error with the **errno** value. The function **strerror** gets the error message, and the **perror** function produces the error message on the standard error stream based on the value of **errno**.

In Windows, when an error occurs, most functions return an error code, usually zero, NULL, or a negative value. Functions can also set an internal error code called the last-error code. The **SetLastError** function sets the last-error code for the calling thread. Information on the last error that occurred can be gotten from the function **GetLastError**. To retrieve the description text for the error in your application, use the **FormatMessage** function with the FORMAT_MESSAGE_FROM_SYSTEM flag. The last-error code is kept in the thread local storage (TLS) so that multiple threads do not overwrite each other's values. The signature of the functions is defined as follows.

```
void SetLastError(DWORD dwErrCode);
DWORD GetLastError(void);
DWORD FormatMessage(DWORD dwFlags, LPCVOID lpSource, DWORD dwMessageId,
DWORD dwLanguageId, LPTSTR lpBuffer, DWORD nSize, va_list* Arguments);
```

Error codes are defined in the WinError.h header file. Error codes are 32-bit values (bit 31 is the most significant bit), as in UNIX. However, the error codes and names are different in UNIX and Windows. UNIX error names begin with an E and Windows error names begin with ERROR_.

Note More information on error codes and error handling in Windows is available at “Debugging and Error Handling” topic on MSDN®.

Signals vs. Events

This section maps the signals in UNIX to their equivalent objects in Windows. In addition, it provides different alternatives for converting UNIX code to Windows code and suggests the pros and cons of each alternative. The following topics are discussed in this section:

- Using native signals in Windows.
- Replacing UNIX signals with Windows messages.
- Replacing UNIX signals with Windows event objects.
- Porting the **Sigaction** call.

This section provides you with information regarding signal-specific implementation in your UNIX application and provides the best replacements, such as native signals, messages, and event objects, for the Windows environment.

The UNIX operating system supports a wide range of signals. UNIX signals are software interrupts that catch or indicate different types of events. Windows, on the other hand, supports only a small set of signals that is restricted to exception events. Consequently, converting UNIX code to Windows requires the use of new techniques to replace the use of some UNIX signals.

The Windows signal implementation is limited to the signals listed in Table 5.1.

Table 5.1. Windows Signals

Signal	Meaning
SIGABRT	Abnormal termination
SIGFPE	Floating-point error
SIGILL	Illegal instruction
SIGINT	CTRL+C signal
SIGSEGV	Illegal storage access
SIGTERM	Termination request

Note When a CTRL+C interrupt occurs, Windows generates a new thread to handle the interrupt. This can cause a single-thread application, such as one ported from UNIX, to become multithreaded, which may result in an unexpected behavior.

The **signal** function allows a process to choose one of the several ways to handle an interrupt signal from the operating system. The **sig** argument is the interrupt to which the signal responds. The argument must be one of the manifest constants defined in SIGNAL.H.

By default, signal terminates the calling program with exit code 3, regardless of the value of **sig**.

When an application uses other signals not supported in Windows, you have the option of using Messages or Events. This section focuses on the Windows mechanisms that you can use to replace some UNIX signals.

Table 5.2 lists the recommended mechanisms that you can use to replace common UNIX signals. Following are the three main mechanisms:

- Native signals
- Messages
- Event objects

Table 5.2. UNIX Signals and Replacement Mechanisms

Signal name	Description	Suggested Replacement on Windows
SIGABRT	Abnormal termination	SIGABRT
SIGALRM	Time-out alarm	SetTimer – WM_TIMER - CreateWaitableTimer
SIGCHLD	Change in status of child	WaitForSingleObject
SIGCONT	Continue stopped process	WaitForSingleObject
SIGFPE	Floating point exception	SIGFPE
SIGHUP	Hang-up	NA
SIGILL	Illegal hardware instruction	SIGILL
SIGINT	Terminal interrupt character	SIGINT
SIGKILL	Termination	WM_QUIT
SIGPIPE	Write to pipe with no readers	WaitForSingleObject

Signal name	Description	Suggested Replacement on Windows
SIGQUIT	Terminal Quit character	WM_CHAR
SIGSEGV	Invalid memory reference	SIGSEGV
SIGSTOP	Stop process	WaitForSingleObject
SIGTERM	Termination	SIGTERM
SIGTSTP	Terminal Stop character	WM_CHAR
SIGTTIN	Background read from control tty	NA
SIGTTOU	Background write to control tty	NA
SIGUSR1	User-defined signal	SendMessage – WM_APP
SIGUSR2	User-defined signal	SendMessage – WM_APP

Note Only POSIX signals are considered in this table. Seventh Edition, System V, and BSD signals have been excluded.

Another mechanism that can be useful when converting some UNIX signals to Windows is event kernel objects.

Using Native Signals in Windows

In the following example, the simple case of catching SIGINT to detect CTRL-C is demonstrated. As you can see from the two examples, support for handling native signals in UNIX and Windows is very similar.

UNIX example: Managing signals

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
/* The intrpt function reacts to the signal passed in the parameter
signum.
This function is called when a signal occurs.
A message is output, then the signal handling for SIGINT is reset
(by default generated by pressing CTRL-C) back to the default behavior.
*/
void intrpt(int signum)
{
printf("I got signal %d\n", signum);
(void) signal(SIGINT, SIG_DFL);
}
/* main intercepts the SIGINT signal generated when Ctrl-C is input.
Otherwise, sits in an infinite loop, printing a message once a second.
*/
int main()
{
(void) signal(SIGINT, intrpt);
while(1) {
```

```
printf("Hello World!\n");
sleep(1);
}
}
```

(Source File: U_ManageSignl-UAMV3C5.01.c)

Windows example: Managing signals

```
#include <windows.h>
#include <signal.h>
#include <stdio.h>

void intrpt(int signum)
{
printf("I got signal %d\n", signum);
(void) signal(SIGINT, SIG_DFL);
}

/* main intercepts the SIGINT signal generated when Ctrl-C is input.
Otherwise, sits in an infinite loop, printing a message once a
second.*/
void main()
{
(void) signal(SIGINT, intrpt);

while(1)
{
printf("Hello World!\n");
Sleep(1000);
}
}
```

(Source File: W_ManageSignl-UAMV3C5.01.c)

Note By default, the signal terminates the calling program with exit code 3, regardless of the value of **sig**. Additional information is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclib/html/_crt_signal.asp.

With the exception of requiring an additional header file and the different signature of the **sleep** function, the two examples are identical.

Replacing UNIX Signals with Windows Messages

UNIX uses signals to send alerts to processes when specific actions occur. A UNIX application uses the **kill** function to activate signals internally. As discussed earlier, Windows provides only limited support for signals. As a result, you must rewrite your code to use another form of event notification in Windows.

The following example illustrates how you can convert UNIX code to Windows messages or event objects. It shows a simple main function that forks a child process, which issues the SIGALRM signal. The parent process catches the signal and outputs a message when it is received.

UNIX example: Using the SIGALRM signal

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

static int alarm_fired = 0;

/* The alrm_bell function simulates an alarm clock. */
void alrm_bell(int sig)
{
alarm_fired = 1;
}

int main()
{
int pid;
/* Child process waits for 5 sec's before sending SIGALRM to its
parent. */
printf("alarm application starting\n");
if((pid = fork()) == 0)
{
sleep(5);
kill(getppid(), SIGALRM);
exit(0);
}
/* Parent process arranges to catch SIGALRM with a call to signal
and then waits for the child process to send SIGALRM. */

printf("waiting for alarm\n");
(void) signal(SIGALRM, alrm_bell);
pause();

if (alarm_fired)
printf("Ring...Ring!\n");
printf("alarm application done\n");
exit(0);
}
```

(Source File: U_SigAlrm-UAMV3C5.01.c)

In the example that follows, a form of Microsoft Windows messages is used to signal the parent process. The **SetTimer** function is used to signal to the parent process that an alarm has been activated. Although code can be created to do the timing, using the **SetTimer** function greatly simplifies this example.

Another advantage of using **SetTimer** is that the callback function is invoked in the same thread that calls **SetTimer**; therefore no synchronization is necessary.

If the requirements are simple, consider using a thread to act as a timer thread that calls **Sleep** to create the desired delay. At the end of the delay, a call is made to a **timer callback** function. The problem with this approach is that the **callback** function is not called from your primary thread. If the **callback** function requires resources that are thread-specific, use one of the appropriate synchronization mechanisms discussed in the "Synchronization of Threads" section in Chapter 3: "Process and Thread Management" of this volume.

Additional code is added to the example so that an application using this code can catch any standard Windows message as well as application and user-defined messages. You can use these messages to engineer solutions to other signals that are not directly supported by the native signal implementation in Windows.

Windows example: Replacing SIGALRM using messages

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

static int alarm_fired = 0;
/* The alrm_bell function simulates an alarm clock. */

VOID CALLBACK alrm_bell(HWND hwnd, UINT uMsg, UINT idEvent, DWORD
dwTime )
{
alarm_fired = 1;
printf("Ring...Ring!\n");
}

void main()
{
printf("alarm application starting\n");
/* Set up a 5 second timer which calls alrm_bell */
SetTimer(0, 0, 5000, (TIMERPROC)alrm_bell);
printf("waiting for alarm\n");
MSG msg = { 0, 0, 0, 0 };
/* Get the message, & dispatch. This causes alrm_bell to be invoked. */

while(!alarm_fired)
if (GetMessage(&msg, 0, 0, 0) )
{
if (msg.message == WM_TIMER)
printf("WM_TIMER\n");
DispatchMessage(&msg);
}
printf("alarm application done\n");
exit(0);
}
```

(Source File: W_SigAlrm-UAMV3C5.01.c)

Notice in this example that the WM_TIMER message is issued by the **SetTimer** function and captured by the **GetMessage** function. If you remove the call to **DispatchMessage**, the **alrm_bell** function is never called, but the WM_TIMER message is received. With this simple application, you can capture a variety of Windows messages. Moreover, if you want to trigger the **callback** function before the specified time, you can use the **PostMessage**(WM_TIMER) call. This is analogous to using the **kill** function to send a signal in UNIX.

Replacing UNIX Signals with Windows Event Objects

Some events that UNIX handles through signals are represented in Windows as event objects. Functions are available to integrate these event objects. An example of these functions is the **WaitForSingleObject** function.

In the example code that follows, a timer object is used to signal when a timed interval has elapsed. Again, this example provides the same functionality as the preceding UNIX SIGALRM example.

Note While this illustration encompasses the process in a single thread, this is not a requirement. The timer object can be tested and waited for in other threads if necessary.

Windows example: Replacing SIGALRM using event objects

```
#define _WIN32_WINNT 0X0500

#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

void main()
{
HANDLE hTimer = NULL;
LARGE_INTEGER liDueTime;
liDueTime.QuadPart = -50000000;
printf("alarm application starting\n");

// Set up a 5 second timer object
hTimer = CreateWaitableTimer(NULL, TRUE, "WaitableTimer");
SetWaitableTimer(hTimer, &liDueTime, 0, NULL, NULL, 0);
// Now wait for the alarm
printf("waiting for alarm\n");

// Wait for the timer object
WaitForSingleObject(hTimer, INFINITE);
printf("Ring...Ring!\n");
printf("alarm application done\n");
exit(0);
}
```

(Source File: W_SigAlrm-UAMV3C5.02.c)

Porting the Sigaction Call

Windows does not support the **sigaction** function. The UNIX example that follows shows how the **sigaction** function is typically used in a UNIX application. In this example, the handler for the SIGALRM signal is set. Conversion of this code to use Windows messages was shown earlier. For the corresponding Windows example, refer to the previous section “Replacing UNIX Signals with Windows Messages.”

Note To terminate the following application from the keyboard, press CTRL+\.

UNIX example: Using sigaction

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

void intrpt(int signum)
{
printf("I got signal %d\n", signum);
}

int main()
{
struct sigaction act;
act.sa_handler = intrpt;
sigemptyset(&act.sa_mask);
act.sa_flags = 0;
sigaction(SIGINT, &act, 0);

while(1)
{
printf("Hello World!\n");
sleep(1);
}
}
```

(Source File: U_SigActn-UAMV3C2.01.c)

Interprocess Communication (IPC)

Like UNIX, Windows has various forms of interprocess communication (IPC). Following are the IPC forms that are most familiar to UNIX developers:

- Process pipes
- Named pipes
- Message queues
- Sockets
- Memory-mapped files
- Shared memory
- Remote procedure call (RPC)

Windows supports implementations of all of these forms except for message queues. Windows provides two additional IPC mechanisms: remote procedure call (RPC) and mailslots. RPC is designed for use by client/server applications and is most appropriate for C and C++ programs. Mailslots are memory-based files that a program can access using standard file functions. The maximum size of mailslots is fairly small.

In addition to these mechanisms, there are two forms of IPC that are not part of the Windows API. These are Message Queuing (also known as MSMQ) and COM+.

This section looks at how UNIX code that uses different forms of IPC can be converted to Windows IPC techniques. It also introduces new methods of IPC that are not available in UNIX but may provide a better solution for interprocess communication of your application. You can use this information to analyze the IPC implementations in your UNIX application and to identify the best porting approach for corresponding IPC implementation in the Windows environment.

Pipes (Unnamed or Named, Half or Full Duplex)

Pipes have similar functionality on both Windows and UNIX systems. Their primary use is to communicate between related processes. UNIX pipes can be named or unnamed. They also have separate read and write file descriptors, which are created through a single function call. With unnamed pipes, a parent process that must communicate with a child process creates a pipe that the child process will inherit and use. Two unrelated processes can use named pipes to communicate.

Windows pipes can also be named or unnamed. A parent process and a child process typically use unnamed pipes to communicate. Unnamed pipes are half duplex. The processes must create two unnamed pipes for bidirectional communication. Two unrelated processes can use named pipes even across the network on different computers. Typically, a server process creates the pipe, and clients connect to the bidirectional pipe to communicate with the server process. Named pipes are full duplex.

Process Pipes

Process pipes are supported in Windows by using the standard C run-time library. As discussed in the following sections, they are largely equivalent to process pipes in UNIX. Three UNIX examples are considered in this section, and a slightly modified Windows version of each is also provided.

High-Level popen Call

Note that this text assumes the presence of the Uname.exe executable program on the Windows-based system. If your system does not contain this executable, these samples will not work, and you must modify them to use an equivalent tool.

In the first example of process pipes, a process called uname is executed and passes the output of this process to standard output. As you review these examples, notice that the differences between the UNIX and Windows implementations are the header files that are required and the function names for **popen** and **pclose**. The names of these functions in Windows are preceded by an underscore (_). The function syntax is the same and the behavior is largely compatible.

Low-Level pipe Call

This section demonstrates the process of converting code that uses the **pipe** function call to communicate between two parts of an application. The following example demonstrates how you can write to one end of the pipe (fd[1]) and read from the other (fd[0]). This is an example of a half duplex pipe. The differences between the UNIX and the Windows implementations are the required header files and the underscore that precedes the **pipe** function.

However, in this case, an additional modification must be made for the solution to work on Windows. If you look at the online documentation for pipe, you will notice that it requires two additional arguments. Providing these arguments specifies the amount of memory that must be used as a buffer for the pipe and the mode of the pipe (O_TEXT or O_BINARY).

UNIX example: Low-level pipe call

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
int data_out, data_in, file_pipes[2];
const char data[] = "ABCDE";
char buffer[BUFSIZ + 1];
memset(buffer, '\0', sizeof(buffer));

if (pipe(file_pipes) == 0)
{
data_out = write(file_pipes[1], data, strlen(data));
printf("Wrote %d bytes\n", data_out);
data_in = read(file_pipes[0], buffer, BUFSIZ);
printf("Read %d bytes: %s\n", data_in, buffer);
exit(EXIT_SUCCESS);
}

exit(EXIT_FAILURE);
}
```

(Source File: U_LowLvlPip-UAMV3C5.01.c)

Windows example: Low-level pipe call

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <process.h>

void main()
{
size_t data_out, data_in;
int file_pipes[2];
const char data[] = "ABCDE";
char buffer[BUFSIZ + 1];
memset(buffer, '\0', sizeof(buffer));

if (_pipe(file_pipes, 32, O_BINARY) == 0)
{
data_out = write(file_pipes[1], data, strlen(data));
printf("Wrote %d bytes\n", data_out);
data_in = read(file_pipes[0], buffer, BUFSIZ);
printf("Read %d bytes: %s\n", data_in, buffer);
exit(EXIT_SUCCESS);
}

exit(EXIT_FAILURE);
}
```

(Source File: W_LowLvlPip-UAMV3C5.01.c)

Named Pipes (FIFOs)

In this section, a few examples of named pipes are shown. These are sometimes referred to as first-in-first-out (FIFO).

Interprocess Communication with Named Pipes

To show how you can convert code using FIFO from UNIX to Windows, a simple example that creates a named pipe is shown. This example uses minimal security restrictions for simplicity. The example uses the **mkfifo** function in UNIX and the **CreateNamedPipe** function in Windows. There are considerable differences between these two functions. Both functions have the same purpose, but the **CreateNamedPipe** function offers a greater degree of control over the configuration of the pipe.

The major difference between UNIX and Windows pipes exists in security attributes, buffering, and opening modes. The **mkfifo**() system call creates a new FIFO file with name path. The access permissions are specified by mode and restricted by the umask routine of the calling process. In the case of Windows pipes, the access control list (ACL) from the security attributes parameter defines the discretionary access control for the named pipe. If SecurityAttributes is NULL, the default security descriptor and the handle cannot be inherited. The ACLs in the default security descriptor for a named pipe grant full control to the LocalSystem account, administrators, the creator owner, and read access to members of the Everyone group and the anonymous account. The details are beyond the scope of this guide, but you can find the links for **CreateNamedPipe** on the MSDN Web site.

Note Additional information on named pipes on Windows is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ipc/base/named_pipe_type_read_and_wait_modes.asp.

Back-Pressure in Pipes

There is an important difference between pipes in UNIX and Windows involving the concept of back-pressure. UNIX uses the buffer concept for dealing with programs that use pipes whereas Windows uses the file system object. This difference becomes a major point to consider when migrating to Windows if the program in question has a tendency to acquire back-pressure. This issue arises in sequences of programs connected by pipes because there is a finite capacity for the pipe in UNIX versus a Windows file system object that will almost never be exhausted, based on page-file size.

Note The point of the guidance is that a set of UNIX applications, which rely on pipe back-pressure to throttle a process, should be reexamined when moved to use native Windows pipes. Instead of relying on back-pressure, a more explicit mechanism should be selected by redesigning the application to remove its dependency on back-pressure.

Shared Memory

Shared memory is an efficient means of passing data between programs. One program will create a memory portion, which other processes (if permitted) can access.

UNIX supports System V shared memory, which allows multiple processes to attach a segment of physical memory to their virtual address spaces. When write access is allowed for more than one process, an outside protocol or mechanism such as a semaphore can be used to prevent inconsistencies and collisions. The other efficient way to implement shared memory in UNIX applications is to rely on the **mmap** function and on the native virtual memory facility of the system. The **mmap** function establishes a mapping of a named file system object (or part of one) into a process address space.

Windows implementation of shared memory can be done using the concept of file mapping or by making use of memory in a heap using the **GlobalAlloc** function. A common section of memory can be mapped into the address space of multiple processes. If no file is specified in the creation function, the shared memory is allocated from a section of the page file. As in the UNIX implementation, which uses an identifier, Windows uses a handle identifier to identify the memory that is mapped into the process at the requested address. The **GlobalAlloc** function allocates the specified number of bytes from the heap. For more information on the global functions, refer to the MSDN Web site.

Both the UNIX and Windows file mapping solutions offer the capability of saving the contents in a permanent file.

Message Queues

The Microsoft Windows API does not support message queues as standard. If you want to use message queuing in your application, you should use Message Queuing (also known as MSMQ). Message Queuing is covered comprehensively in other Microsoft documentation and is therefore only briefly described here.

Message Queuing technology enables applications running at different times to communicate across heterogeneous networks and systems that may be temporarily offline. Applications send messages to queues and read messages from queues. Message Queuing provides guaranteed message delivery, efficient routing, security, and priority-based messaging. It can be used to implement solutions for both asynchronous and synchronous scenarios that require high performance.

Note Additional information on Message Queuing is available at http://www.microsoft.com/windows2000/technologies/communications/msmq/default.mspx.

Networking

The primary networking protocol for UNIX and Windows is TCP/IP. The standard programming API for TCP/IP is called sockets. The Windows implementation of sockets is Winsock 2 (formerly known as Windows Sockets). Winsock conforms well to the Berkeley implementation, even at the API level. Most of the functions are the same, but slight differences in parameter lists and return values do exist. RPC is another networking concept. Microsoft RPC provides RPC mechanisms for Windows-based applications. The following sections describe sockets and RPC mechanisms in UNIX and Windows. With this knowledge, you will be able to analyze UNIX networking applications and identify the appropriate porting approach for corresponding sockets and RPC implementations in the Windows environment.

TCP/IP and Sockets

Winsock 2 uses the sockets paradigm that was first popularized by Berkeley Software Distribution (BSD) UNIX. It was later adapted for Microsoft Windows in Winsock 1.1. One of the primary goals of Winsock has been to provide a protocol-independent interface that is fully capable of supporting emerging networking capabilities, such as real-time multimedia communications.

Winsock is an interface, not a protocol. As an interface, it is used to discover and use the communications capabilities of any number of underlying transport protocols. Because it is not a protocol, it does not in any way affect the bits on the wire and does not need to be used on both ends of a communications link.

Winsock programming previously centered on TCP/IP. Some of the programming practices that worked with TCP/IP do not work with every protocol. As a result, the Winsock API added new functions that were necessary to handle several protocols.

Winsock has changed its architecture to provide easier access to multiple transport protocols. Following the Windows Open System Architecture (WOSA) model, Winsock now defines a standard service provider interface (SPI) between the API with its functions exported from Ws2_32.dll and the protocol stacks. Consequently, Winsock support is not limited to TCP/IP protocol stacks as is the case for Windows Sockets 1.1. More information is available at the discussion on "Windows Sockets 2 Architecture" in Microsoft Visual Studio® .NET 2003.

There are new challenges in developing Winsock 2 applications. When sockets only supported TCP/IP, a developer could create an application that supported only two socket types: connectionless and connection-oriented. Connectionless protocols use SOCK_DGRAM sockets, and connection-oriented protocols use SOCK_STREAM sockets.

Because of the many new socket types introduced, developers can no longer rely on the socket type to describe all the essential attributes of a transport protocol.

Sockets are not a part of the Windows API. You will need to consult the Platform SDK for detailed information about the Winsock APIs. The help for the Platform SDK contains complete samples that demonstrate how to implement socket-based client and server applications. For an in-depth comparison between UNIX sockets and Winsock, refer to the discussion of "Porting Socket Applications to Winsock" in the Microsoft Platform SDK.

Note More information is available in the “Write Scalable Winsock Apps Using Completion Ports” MSDN article at http://msdn.microsoft.com/library/default.asp?url=/msdnmag/issues/1000/Winsock/toc.asp.

Remote Procedure Calls

In UNIX, routines that allow C programs to make remote procedure calls (RPCs) are in the rpc/rpc.h library.

In Windows, RPC for the C and C++ programming languages is provided by the Microsoft RPC. Microsoft RPC is compatible with Open Software Foundation (OSF) Distributed Computing Environment (DCE) RPC. Microsoft RPC differs from DCE RPC in the location service, which Microsoft RPC does not offer. When you have located an RPC service, you have binary compatibility. Microsoft RPC fully supports 64-bit Windows. Using RPC, developers can transparently communicate between different types of processes. RPC automatically manages process differences behind the scenes.

The process for creating a client/server application using Microsoft RPC is:

1. Develop the interface.
2. Develop the server that implements the interface.
3. Develop the client that uses the interface.

All interfaces for programs using RPC must be defined in Microsoft Interface Definition Language (MIDL) and compiled with the MIDL compiler.

Windows Server 2003 Features

RPC includes a collection of new capabilities and improvements in Microsoft Windows Server™ 2003. Some of these capabilities are available in Windows XP as well, but are new for the server family of Windows.

The following capabilities are new in Windows Server 2003:

- **NDR64.** NDR64 is a new transfer syntax optimized for 64-bit environments.
- **RPC over HTTP**. RPC now enables its clients to securely and efficiently connect across the Internet to RPC server programs and execute remote procedure calls. More information is available in “Remote Procedure Calls Using RPC over HTTP” on the MSDN Web site.
- **Improved troubleshooting.** RPC has improved its troubleshooting capabilities and extended error information. Refer to “Obtaining Extended RPC Error Information” on the MSDN Web site.

Other new capabilities include better configuration using RPC NetShell extensions, improved SDK samples such as FileRep, and support for RPC verifier to automatically check for errors in RPC code.

Note Guidance can be found on the MSDN Web site at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/rpc/rpc/best_rpc_programming_practices.asp.

Also, refer to the following URL at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/rpc/rpc/overviews.asp.

Miscellaneous Features

This section focuses on the miscellaneous infrastructure areas of the UNIX and Windows environments, listed as follows:

- Shells and scripting
- Scripting languages
- Daemons versus services
- Middleware

These topics give you information regarding how you can replace these specific implementations in your UNIX application with the equivalent features of the Windows environment.

Shells and Scripting

A shell is a command-line interpreter that accepts typed commands from a user and executes them. In addition to executing programs, shells usually support advanced features such as the capability to recall recent commands and a built-in scripting language for writing programs.

Programs written through the programming features of a shell are called shell scripts. In addition to scripts written through the use of shells, there are also languages specifically designed for writing scripts. As with shell scripts, these scripting languages are interpreted.

The use of scripting languages leads to rapid development, often with relaxed syntax checking, but slower performance. Windows and UNIX support a number of shells and scripting languages, some of which are common to both operating systems.

Command-Line Shells

On the Windows platform, **Cmd.exe** is the command prompt, or the shell. With the command prompt, a user can run programs or scripts and invoke applications. The command prompt has a memory or buffer for recent commands, so the user can retrieve, run, and edit them using various techniques.

On UNIX, a number of standard shells provide the UNIX user interface. These shells include:

- **The Bourne shell (sh)**. This is the simplest shell, often set as the default. It can invoke programs and create pipes, but it has no command memory or advanced scripting capabilities.
- **The C shell (csh)**. This shell includes command memory and a scripting language similar to the C language. An Interix/UNIX version of the C shell comes with the Interix product.
- **The Korn shell (ksh)**. The Korn shell also features command memory and a built-in language for creating script files. The Korn shell is based on the Bourne shell but includes additional features, such as job control, command-line editing, functions, and aliases. Interix/UNIX versions of the Korn shell are delivered with Windows Services for UNIX and Interix products.

Scripting Languages

The following sections explain the scripting languages and scripting-language support provided in Windows and UNIX.

- **Windows Script Host**. Windows Script Host (WSH) is a language-independent environment for running scripts and is often used to automate administrative tasks and logon scripts. WSH provides objects and services for scripts, establishes security, and invokes the appropriate script engine, depending on script language. Objects and services supplied allow the script to perform such tasks as displaying messages on the screen, creating objects, accessing network resources, and modifying environment variables and registry keys. WSH natively supports Microsoft Visual Basic® Scripting Edition (VBScript) and Microsoft JScript®. Other languages that are available for this environment are Perl, REXX, and Python. WSH is built in to all versions of Windows after Microsoft Windows® 95. It can also be downloaded or upgraded from the Microsoft Web site.
- **Perl**. Perl is an acronym for Practical Extraction and Report Language. It is an interpreted language that was originally designed for UNIX, but it has since been ported to many platforms. Perl provides a cross-platform scripting environment that developers can use to write scripts that can be run on both Windows and UNIX. Perl is effective for string manipulation. Although Perl is not delivered with Windows, there are many versions of Perl available that are designed to run on Windows. Perl also comes with Windows Services for UNIX.
- **REXX**. REXX is an acronym for Restructured Extended Executor Language and was originally developed by IBM United Kingdom Laboratories. It is a procedural language that is designed for application programs to use as a macro or scripting language. Although REXX can issue commands to its host environment and can call programs and functions written in other languages, it is designed to be independent of a specific operating system. Versions of REXX are available for both UNIX and Windows.
- **Python**. Python, just like Perl, is an interpreted language. Many of its features are similar to Perl, but its programming structure and syntax are clearer, thus making Python code easier to read and maintain. Although it was designed for UNIX, it is now widely available on other platforms, including Windows. Python is object-oriented and includes dynamic data structures and dynamic typing. Python is ideal for rapid software development where maintainable code is important. Python is not shipped with Windows, but it can be downloaded from the Python Web site at http://www.python.org/download/.
- **Tcl/Tk**. Tcl/Tk is yet another interpreted language. Like Perl, it is effective for string manipulation and is available across UNIX and Windows platforms. Tcl/Tk is particularly applicable to the development of cross-platform GUIs. Tcl/Tk is not shipped with Windows, but it can be downloaded from the Tcl/Tk Web site at http://www.tcl.tk.

Daemons vs. Services

A UNIX daemon is a process that runs in the background and does not require a user interface. A service application is the equivalent of a daemon on Windows. Normally, a daemon is started when the system is booted and runs without supervision until the system is shut down. Similarly, a Windows service can be started at boot time and run until system shutdown. However, the Service Control Manager (SCM) controls all services. To convert daemon code to run on Windows, you must add code to interface with SCM. This section briefs you on the Windows API functions to convert UNIX daemons to Windows services.

Unless the **main** function of the daemon is extremely simple, the best strategy is to rename it to something else, such as *service_main***.** Then create a new main that contains code to install, uninstall, and run the service depending on command-line arguments.

To install the program as a service, the program must call **OpenSCManager** to get a handle to the SCM. It must then call **CreateService**, passing the SCM handle and several arguments, including the service name, display name, service type, and path to the executable, and identity the service uses to run. After the service is installed, the administrative tools services applet can be used to examine and modify many of these values.

Uninstalling the service is similar to installing it. Call **OpenSCManager** to get a handle to the SCM, and then call **OpenService** to get a handle to the service. If it is running, the service should be stopped by calling the **ControlService** function. Finally, call **DeleteService**, passing the service handle, and close the handles to clean up.

Running the daemon as a service is also fairly straightforward. The new main function sets up a SERVICE_TABLE_ENTRY structure that contains a name and a pointer to the main function of the service, which is the old main function renamed *service_main*. This structure is passed to the **StartServiceCtrlDispatcher** function, which does not return until the service stops. Note that more than one service entry can be present in the service entry structure, so a single executable can support more than one service. The definitions are the same as a **main** function; however, the arguments to the main function of the service are supplied by SCM and can be set through the services applet or the **CreateService** call.

The main function of the service needs new code to call the **SetServiceStatus** function, which keeps SCM informed of the status of the service during startup. If the SCM does not receive status updates within a specified time period, it assumes that the service has stopped running and logs an error. The SCM must also be given the address of a service control function that the SCM can use to request actions such as requesting the service to stop.

Call the **RegisterServiceCtrlHandler** or **RegisterServiceCtrlHandlerEx** function to set this address. When the service is fully initialized, it should call **SetServiceStatus** with the SERVICE_RUNNING status to complete the startup sequence.

Note For sample service programs and details of the service functions, refer to the MSDN Web site at http://msdn.microsoft.com/.

Middleware

This section compares the various middleware solutions available for UNIX-based and Windows-based applications. With the information provided in this section, you will be able to identify the various middleware technologies available on UNIX and the alternative Windows technologies to use in migrating UNIX middleware to the Windows environment.

OLTP Systems

Online transaction processing (OLTP) systems have been implemented in UNIX environments for many years. These systems perform such functions as resource management, threading, and distributed transaction management. OLTP systems typically provide support for multiple languages and development environments.

Common OLTP systems include:

- Tuxedo from BEA Systems.
- TOP END from NCR Corporation.
- Encina for DCE (distributed computing environment) from Transarc.

Although OLTP was originally developed for UNIX, many OLTP systems also have Windows versions. Additionally, gateways exist to integrate systems that use different transaction monitors—for example, the Tuxedo gateway to Top End. OLTP systems currently face the challenge of integrating with Web and e-business systems. Many OLTP systems provide a bridge to the Java programming language and provide gateways to Common Object Request Broker Architecture (CORBA) and COM.

When considering transaction and resource management during a UNIX migration, developers should remember that OLTP systems provide many of the same features as COM+. As with most cross-platform products, OLTP monitors achieve these features by introducing new APIs to the development environment. Introducing COM+ for transaction and resource management during a migration can lessen this type of dependency.

Queuing Systems

Message queuing is provided as a feature in AT&T System V UNIX and can be achieved through sockets in Berkeley UNIX versions. These types of memory queues are most often used for interprocess communications and do not meet the requirements for persistent store and forward messaging.

To help meet these requirements in UNIX, versions of MQSeries from IBM and MessageQ (formally the DEC MessageQ) from BEA Systems are available. A reliable and resilient store-and-forward message queue provides a key building block for enterprise integration and highly available, loosely coupled systems.

Microsoft provides similar functionality for Windows through Message Queuing. IBM and BEA Systems also provide versions of their queuing systems for Windows. Gateway offers products that bridge the various queuing systems. One reason for migrating to Windows may be the need to integrate with commercial off-the-shelf applications. The queuing system for such a migration would need to provide an API that easily integrates into these applications. For example, Message Queuing provides for a COM Automation Interface API and .NET classes.

Component-based Development in Windows

The Windows platform offers developers a wide range of component-based development tools and technologies. One of these is the Component Object Model (COM).

Component Object Model

COM is the first component-based development technology from Microsoft. Developers can use COM to develop component-based software by exploiting a set of well-defined development techniques and run-time services. By adhering to the COM development model and by using one of the many COM-aware development environments, developers can easily build component-based software that is capable of interacting with other components developed by different organizations, potentially in different development languages.

Although many of the required development techniques—such as how functionality should be exposed through interfaces—are complex, the development environments available on the Windows platform mask this complexity. One of the most popular development environments is Visual Basic.

Some of the key features of the COM programming model are as follows:

- COM objects expose functionality through well-defined interfaces, the binary format of which is defined by the COM specification. This functionality matches the classic C++ virtual function table [v-table] layout in memory.
- An interface consists of a set of methods. However, most development environments also allow properties to be exposed at the interface level through a pair of property-get and property-set methods.
- COM supports component versioning.
- COM components can be hosted in-process (through DLLs), out-of-process (through executable files), or in executable files on remote computers.
- All COM components and COM interfaces on a particular computer are logged centrally in the Windows registry, which is a hierarchical configuration database for the Windows platform.

The Windows registry contains such information as system hardware details, hardware and system configuration, and details of applications installed on the system. For COM, the registry stores a globally unique identifier (GUID) to identify each component class and interface installed. GUIDs are 128-bit integers that are guaranteed to be unique. COM uses this information to determine which component class to create when an application make a request for an object (component) to be instantiated.

Each component also has a user-friendly name known as a ProgID, or programmatic identifier, that is created by the component vendor. The ProgID is not guaranteed to be unique. The recommended format for a ProgID is *vendor.component.version*, where vendor and component are alphanumeric names.

When an application needs to use an object, it starts by calling the **CoCreateInstance** COM function to create the component. This function takes the registered GUID for the object class (CLSID) as an argument. If the developer chooses to use the user-friendly ProgID instead, the application first calls a function to get the CLSID from the ProgID. The application may also pass the initial interface GUID to **CoCreateInstance**, or it may pass a null entry to receive the default interface. COM finds the server for the class, loads the class into memory if necessary, and marshals the call if the server is in another process or across the network.

After a COM component is created, the application can query a particular interface and use the interface to perform work. Because the interfaces are identified by GUIDs just as the components are, the **QueryInterface** call takes the GUID as an argument and either returns the interface requested or returns a null entry if the interface is not implemented by the class.

Note Additional information about COM is available at
http://www.microsoft.com/com.

Chapter 6: Developing Phase: Migrating the User Interface

This chapter describes how to migrate from a UNIX-based user interface to a Microsoft® Windows® user interface. Because the majority of UNIX graphical interfaces are built on X Windows and Motif, the chapter focuses on porting code from X Windows to the Windows operating system.

This chapter describes:

- The architectural and visual differences between the two environments.
- The programming principles used by X Windows and Windows.
- How to migrate each type of graphical construct from one environment to the other.

With this knowledge, you will be able to choose the most suitable methodology for migrating your applications from the UNIX user interface to the Windows user interface. You will also be able to map between X Windows UI routines and Windows UI routines in order to migrate X Windows applications.

Comparing X Windows with Win32/Win64 GUI

This section describes the difference in architectures between X Windows and the Microsoft Win32®/Win64 GUI. The main user interface type in use on the UNIX platform today is built on the X Windows set of standards, protocols, and libraries. When migrating such a user interface, it is important to compare the user interface architecture and the resulting “look and feel” of the two models. It is also useful to understand the differences in user interface terminology between the two environments.

User Interface Architecture

The X Windows-based interfaces architecture differs from that of Windows. The first and most fundamental difference is the orientation of client and server. For X Windows, the client is the application that requests services and receives information from the user interface.

The user-facing elements of the interface are based on what is termed the X Server. In the X Windows-based system, the client application sends requests to the server to display graphics and to send mouse and keyboard events. The X Server is responsible for doing all the work on behalf of the client. The client can run on a remote system with no graphics hardware or on the same physical computer as the server. In either case, the client does not interact with the display, mouse, or keyboard. This is shown in Figure 6.1, which represents the X Windows client/server architecture.

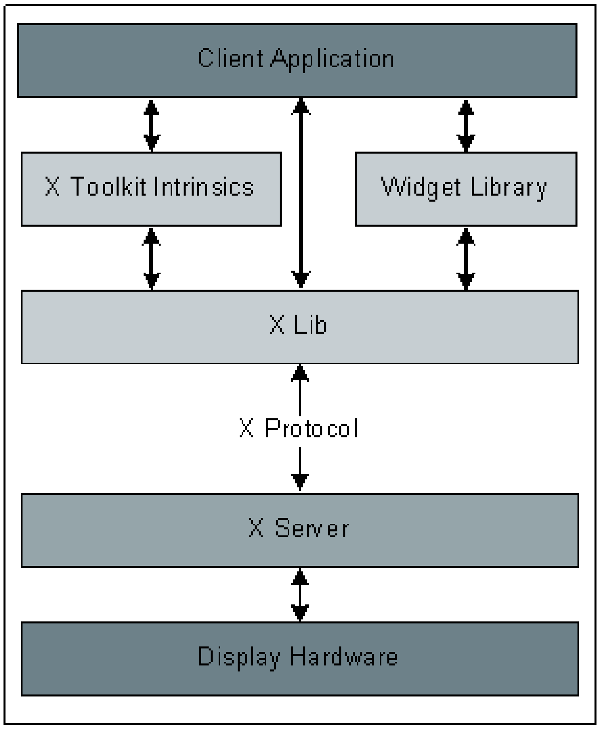


Figure 6.1. The X Windows architectural model

The Windows environment is implemented as a client/server system. Applications are bound to the application programming interface (API) exported by the operating system by link-time binding.

The environment is the server and the Windows API programs are the clients. The Windows API application links to the client-side dynamic-link libraries (DLLs), such as GDI32.dll or user32.dll. When the application makes any Graphics Device Interface (GDI) function calls, the client-side GDI DLL will invoke the native system call in the kernel mode. This is because the GDI components reside in the kernel mode. This results in a switch from the user mode to kernel mode and does not require any kind of message passing and context switching, thereby improving performance.

Figure 6.2 illustrates the Windows architectural model.

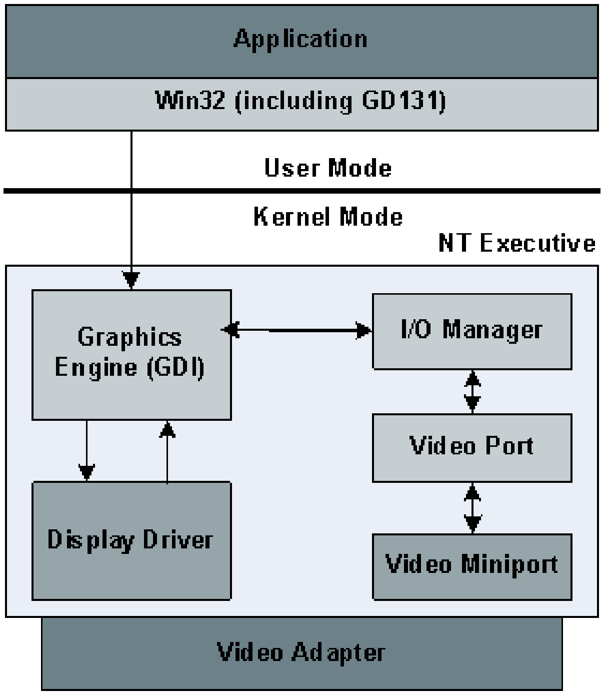


Figure 6.2. The Microsoft Windows UI architectural model

Elements of the UI

Windows types are very similar in both the X Windows and Windows environments, as detailed as follows:

- **Application window**. The application window is the interface between the user and the application. Elements such as a menu bar, status bar, window menu, minimize and maximize buttons, close button, title bar, sizing border, client area, and scroll bars typically appear in the application window.
- **Dialog boxes**. A user typically accesses a dialog box as a temporary window used to create some additional input. A dialog box contains one or more controls, such as buttons and check boxes, to extract user input. Windows API programs can be entirely dialog-based just as in UNIX.
- **Controls**. Controls such as X Windows widgets come in all shapes, sizes, colors, and functions. There are two ways to create controls in a Windows API environment. One is by using the resource editor, where you can drag and drop the controls from the tool bar onto the window because controls are also windows. The other method of creating a control is by using the **CreateWindow** API by specifying the required attributes.
- **Property sheets**. Property sheets are tabbed dialog boxes. They can be used to select a number of settings for a particular application.

User Interface Programming in X Windows and Microsoft Windows

The basics of getting a Windows-based application and an X Windows/Motif-based application started are similar conceptually. Also, the libraries and core functions are similar in both environments. This section discusses the programming principles for developing user interfaces in the two environments. Using the information provided in this section, you will be able to choose the appropriate technology for migrating from the X Windows UI to the Windows UI and understand the Windows UI programming concepts.

Programming for Windows

When programming for Windows, you can use either the C style API or the C++ style API. As far as the C++ style API is concerned, the programming approach is object-oriented programming.

Note X Servers are available for Windows. These run Win32/Win64 unmanaged code programs. Third-party tools, such as MKS and Hummingbird XDK, are available to run X Windows applications in Win32.

To program in C++ on Windows, the following libraries are available:

- Microsoft Foundation Classes (MFC)
- Active Template Library (ATL)
- GDI+
- .NET Languages

Microsoft Foundation Classes (MFC)

In Visual C++®, Microsoft provides a class library called Microsoft Foundation Classes (MFC) with around 200 classes. The classes in MFC provide a wrapper around the Windows API, thereby providing the user with a set of object-oriented programming (OOP) tools for Windows programming. MFC encapsulates the Windows API. MFC provides various sets of classes for the following categories:

- MFC application architecture classes
- Window support classes
- Drawing and printing classes
- Simple datatype classes
- Collection classes
- File and database classes
- Internet and networking classes
- OLE Linking and Embedding classes
- Debugging and exception classes

Active Template Library (ATL)

Active Template Library (ATL) is known primarily for its COM support. It also provides several classes that simplify Microsoft Windows programming. Like the rest of ATL, these classes are template-based C++ classes and have very low overhead. ATL can be used to create windows and dialog boxes and to handle messages.

GDI+

GDI+ is a class-based API for C/C++ programmers. It enables applications to use graphics and formatted text on video and on print. Windows API-based applications do not access graphics hardware directly. Instead, GDI+ interacts with device drivers on behalf of applications. GDI+ is also supported by the 64-bit Windows operating system.

To use GDI+, the developer must copy the Gdiplus.dll library to the system directory of the user’s computer. For information about the operating systems required in order to use a particular class or method, refer to the “Requirements” section of the documentation for the class or method.

All Windows-based applications can use GDI+, a new technology in the Microsoft Windows XP and Windows Server™ 2003 operating systems. It is also available for applications that run on older operating systems, such as Microsoft Windows NT® 4.0 SP6, Windows 98, Windows 2000, and Windows Millennium Edition operating systems.

Note You can download the latest redistributable at http://www.microsoft.com/downloads/search.aspx?displaylang=en.

.NET Languages

Microsoft .NET is the latest component-based development platform from Microsoft. From a high-level perspective, .NET facilitates component-based development in addition to radically extending the development platform and providing the tools and technologies that developers can use to develop a new kind of Internet-based distributed application.

Note For more information, refer to Volume 4, *Migrate Using .NET* of this guide.

Choosing the Programming Language

When developing components and applications, you can choose between any of the approaches described in this section.

- **Using ATL**. ATL is a fast and easy way to create a COM component in C++ and to maintain a small footprint. ATL can be used to create a control that does not need all of the built-in functionality that MFC automatically provides.
- **Using MFC**. MFC allows you to create full applications, ActiveX controls, and active documents. If you have created a control with MFC, you may want to continue development in MFC. When creating a new control, consider using ATL if you do not need all of the built-in functionality of MFC.
- **Using .NET languages**. This is a good choice in situations where modules are being developed in different .NET languages and they need to communicate with each other without any marshalling, as is needed in the case of COM. For more information, refer to Volume 4, *Migrate Using .NET* of this guide.

In the remainder of this chapter, examples will be given in both the C style API as well as the C++ style API.

Programming Principles

The basic structure of an X Windows-based application that uses Motif is quite similar to the structure of a Windows-based application.

To initiate an X Windows-based interface

1. Initialize the toolkit.
2. Create widgets.
3. Manage widgets.
4. Set up callbacks.
5. Display widgets.
6. Enter the main program event handler.

The following code illustrates these steps:

```
topWidget = XtVaAppInitialize();
frame = XtVaCreateManagedwidget("frame",xmFrameWidgetFrams,
topWidget,,,);
button = XmCreatePushButton( frame, "EXIT", NULL, 0 );
XtManageChild(button)
XtAddCallback( button, XmNactivateCallback, myCallback, NULL );
XtRealizeWidget( topWidget );
XtAppMainLoop();
```

To initiate a Microsoft Windows-based application

1. Initialize an instance and register the Window class.
2. Set up callbacks.
3. Create the window.
4. Display the window.
5. Enter the main program message loop.

The following C style API code illustrates these steps:

```
windowClassStruct.hInstance = thisApplicationInstance;
windowClassStruct.lpfnWndProc = (WNDPROC)myCallback;
RegisterClass( &windowClassStruct );
myWindow = CreateWindow();
ShowWindow(myWindow);
while( GetMessage(,,,) ) {…}
```

X Windows clients and Windows API-based applications rely on events and messages from the outside to manage input devices. The X Windows and Windows APIs use a similar method for this: a message loop where a **callback** function or inline code executes based on the nature of the event or message.

The following code shows a simple Windows API message loop:

```
while ( GetMessage( &msg, NULL, 0, 0 ) ) {
TranslateMessage( &msg );
DispatchMessage( &msg );
}
```

The **GetMessage** API returns an MSG structure (&msg). Right now, only the **message** member of this structure (UINT message in the following example) is of interest. Windows places the message identifier in this field. The developer can use this in the message loop to capture device events.

```
typedef struct tagMSG {
HWND hwnd;
UINT message;
WPARAM wParam;
LPARAM lParam;
DWORD time;
POINT py;
} MSG, *PMSG;
```

In ATL, the event handling is implemented with the MESSAGE MAP macros defined within a window class.

To make use of the message handling feature in the CWindowImpl ATL class

1. Define a class CMyWindow derived from CWindowImpl.
2. Define the Message Map.
3. Define the member functions of **CMyWindow** that can handle the messages.
4. Create an instance of **CMyWindow** class.
5. Create the Window using the **Createmember** function of the **CWindowImpl** class.

```
// Step1 Define the new class and the new class's name must be passed
as an
// argument to the CWindowImpl template.
class CMyWindow : public CWindowImpl<CMyWindow>
{
//Note
//Step 2 Define the Message Map
BEGIN_MSG_MAP(CMyWindow)
// Implies that the OnPaint member function
//will be invoked for the message WM_PAINT
  MESSAGE_HANDLER(WM_PAINT,OnPaint)
  MESSAGE_HANDLER(WM_CREATE,OnCreate)
  MESSAGE_HANDLER(WM_DESTROY,OnDestroy)
END_MSG_MAP()
//Step3 Define the member functions that handle the messages:

LRESULT OnPaint(
  UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled )
{ ...
}
LRESULT OnCreate(
  UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled )
{ ...
}
LRESULT OnDestroy(
  UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled )
{ ...
}
}; // CMyWindow
……
//Step 4 Create an object of class CMyWindow
CMyWindow wnd;   // constructs a CMyWindow object
//Step5 Create a window on the screen
wnd.Create( NULL, CWindow::rcDefault, _T("Hello"),
  WS_OVERLAPPEDWINDOW|WS_VISIBLE );
```

Detailed information on message maps in ATL is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcmfc98/html/_atl_message_maps.asp.

Note IA64

LPARAM, WPARAM, and LRESULT are the typical polymorphic types; they hold pointers or integral types. When returning data from wParam and lParam fields of the MSG type variable, do not assign these values to variables of data types DWORD, ULONG, UINT, INT, int, or long values.

In this case, the Win64 compiler will give the following warning:

Warning C4244: 'return': conversion from 'WPARAM' to 'int', possible loss of data

Creating Windows

The code examples in this section show X Windows and Windows API implementations of window management. It is not likely that any large-scale X Windows client or Windows API-based application would actually be implemented as these short code examples. However, it is easy to see the conceptual similarities and some differences as well.

An X Windows X11 client might use the **XtAppInitialize, XtVaAppInitialize, XtOpenApplication,** or **XtVaOpenApplication** function to get a top-level widget to create a window, as shown in the following example code.

```
main (int argc, char *argv[] )
{
Widget toplevel; /* Conceptual Application Window */
XtAppContext app; /* context of the app */
toplevel = XtVaAppInitialize( &app,
"myClassName",
NULL,0,&argc,argv,NULL,NULL );
OR
toplevel = XtOpenApplication( &app,
"myClassName",
NULL,0,&argc,argv,NULL,
whateverWidgetClass, NULL,0);
      }
```

In the following code, a Windows API-based graphical application creates a main window.

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
WNDCLASS wndclass;
   wndclass.style =           CS_HREDRAW | CS_VREDRAW;
   wndclass.lpfnWndProc =     WndProc;
   wndclass.hInstance =       hInstance;
   wndclass.hIcon =     LoadIcon(NULL,    IDI_APPLICATION);
   wndclass.hCursor =   LoadCursor(NULL, IDC_ARROW);
   wndclass.hbrBackground = GetStockObject(WHITE_BRUSH);
   wndclass.lpszMenuName = NULL;
   wndclass.lpszClassName = " myClassName ";

   RegisterClass(&wndclass);


HWND hWnd; // handle to the Application Window
hWnd = CreateWindow( "myClassName",
"myWindowsName",
WS_OVERLAPPEDWINDOW,
CW_USEDEFAULT,
0,
CW_USEDEFAULT,
0,
NULL,
NULL,
hInstance, // context of the app
NULL);
```

Note IA64

When setting the **cbWndExtra** member of the **WNDCLASS** structure, be sure to reserve enough space for pointers.

User-exposed, named kernel objects such as HWND are 32 bits only. Ensure that explicit values of HANDLE are not used. For example, use INVALID_HANDLE_VALUE instead of 0xffffffff.

In ATL, the class CWindow can be used to create the window as shown in the following code.

```
CWindow win;
win.Create( "button", NULL, CWindow::rcDefault, "Click me",
  WS_CHILD );
win.ShowWindow( nCmdShow );
win.UpdateWindow();
```

Common Dialog Boxes

The Common Dialog Box Library contains a set of dialog boxes for performing common tasks, such as opening files and printing documents. The common dialog boxes provide a uniform user interface that lets users carry out these common tasks in the same way for each application, which makes for a rapid migration of the UI.

The common dialog boxes include:

- The **Open** and **Save As** file dialog boxes.
- The **Find** and **Replace** editing dialog boxes.
- The **Print**, **Print Setup**, **Print Property Sheet**, and **Page Setup** printing dialog boxes.
- The **Color** and **Font** dialog boxes.

Note Additional information on the Common Dialog Box Library is available at http://msdn.microsoft.com/library/en-us/winui/WinUI/WindowsUserInterface/UserInput/CommonDialogBoxLibrary.asp.

Creating Dialogs Boxes

The following section describes different types of dialog boxes and their implementation. Dialog boxes can be of either modal or modeless.

Modeless Dialog Box

When the system creates a modeless dialog box, it becomes the active window. The modeless dialog box does not disable its parent window or send messages to its parent window. However, it stays at the top of the z-order even if its parent window becomes the active window.

Applications can create a modeless dialog box by using the **CreateDialog** function, with arguments to specify the identifier of a dialog box template and the pointer to the callback procedure that handles messages for the window. The main characteristic of the modeless dialog boxes is that they allow the event loop of their parent to continue processing messages while they operate. For example, a progress dialog that displays progress indicators of processing done by the parent.

Modal Dialog Box

A modal dialog box becomes the active window when the system creates it. Until a call to **EndDialog,** the dialog box remains the active window. Neither the application nor the user can make the parent window active before calling **EndDialog**. An application uses the **DialogBox** function with a resource identifier to create a modal dialog box. Use a modal dialog box when it is desirable to force user input before proceeding. All common dialog boxes are modal except the **Find** and **Replace** dialog boxes.

A dialog box procedure is similar to a window procedure in that the system sends messages to the procedure when it has information to give or tasks to carry out. Unlike a window procedure, a dialog box procedure never calls the **DefWindowProc** function. Instead, it returns TRUE if it processes a message or FALSE if it does not.

The following is a sample dialog box procedure:

```
INT_PTR CALLBACK MyDialogProc(HWND hWnd, UINT uMsg, WPARAM wParam,
LPARAM lParam)
{
switch (uMsg)
{
case WM_COMMAND:
switch (wCommand)
{
case IDOK:

hList = GetDlgItem(hWnd, IDC_LISTNAMES);
GetDlgItemText(hWnd, IDC_TEXT, buffer, 10);
SendMessage(hList,LB_ADDSTRING,i_count,(LPARAM) buffer);
MessageBox(hWnd, buffer, _T("Names"), MB_OK);

return TRUE;
}
case WM_CLOSE:
{
 EndDialog(hWnd, 0);
 return TRUE;
}
      return FALSE;
}
```

Note IA64

To prepare the code for Win64, the DialogBox Procedure return type must be INT_PTR. If the return type is BOOL, the compiler throws an error while creating the dialog. This is because, on the Windows API, the return types BOOL and INT_PTR are of the same size, whereas on Win64 the INT_PTR return type size is 64 bits.

If you use the LRESULT as the return type, both Win32 and Win64 display no warning because the LRESULT and LONG_PTR result types map to each other. The INT_PTR and LONG_PTR result types are of the same size on both x86 and Itanium.

The following **CreateDialog** function creates the dialog box and all the controls that it contains. Note that the second parameter is the name of the dialog box as it appears in the first line of the resource text.

```
/*
** Create modeless dialog box.
*/
hExampleDlg = CreateDialog( hInstance,
MAKEINTRESOURCE(IDD_DIALOG1),
(HWND)NULL,
MyDlgProc );
```

The following **DialogBoxParam** function creates a modal dialog box and all the controls that it contains, once again from a dialog box template resource.

```
/*
** Create modal dialog box.
*/
int value = 1;
DialogBoxParam(GetModuleHandle(NULL),
MAKEINTRESOURCE(IDD_DIALOG1),
(HWND)NULL,
MyDialogProc,
value);
```

Note IA64

The last parameter of the **DialogBoxParam** function is an IN parameter that specifies the value to pass to the dialog box in the LPARAM parameter of the WM_INITDIALOG message.

It is acceptable to have an integer variable in place of LPARAM because the conversion does not result in any loss of data.

EndDialog must be invoked to destroy a modal dialog box, and **DestroyWindow** must be invoked to close a modeless dialog box. The following sample shows the usage of the DestroyWindow function.

```
  LRESULT OnClose( UINT, WPARAM, LPARAM, BOOL& )
      {
           DestroyWindow();
       return 0;
      }
 …..
};
```

In the case of ATL, you can create dialog boxes using the ATL class **CDialogImpl**.

The following C++ code example creates a dialog box by defining a new class **CMyDialog** derived from the ATL class **CDialogImpl.**

IDD_DIALOG1 is the dialog template resource identifier. WM_INITDIALOG is the event to be handled to perform any initialization before the dialog box is displayed. In the following code example, this is done in the event handler **OnInitDialog,** which is a member function of the **CMyDialog** class.

```
class CMyDialog: public CDialogImpl<CMyDialog>
{
public:
  enum { IDD = IDD_DIALOG1 };
  BEGIN_MSG_MAP( CMyDialog )
   MESSAGE_HANDLER( WM_INITDIALOG, OnInitDialog )
  END_MSG_MAP()

  LRESULT OnInitDialog( UINT, WPARAM, LPARAM, BOOL& )
  {
   //Initialize here
```

```
      ….
   return 0;
  }

  };
```

The **CDialogImpl** class can be used to create both modal as well as modeless dialog boxes. The following code illustrates how to display a modal dialog box:

```
STDMETHODIMP CMyApplication::InfoBoxModal()
{
  CMyDialog dlg;
  dlg.DoModal(); // Displays a modal Dialog box
  return S_OK;
}
```

The following code displays a modeless dialog using the **Create** member function, which takes as parameter the object representing the parent window.

```
int APIENTRY WinMain( HINSTANCE hInstance, HINSTANCE, LPSTR, int )
{
  _Module.Init( NULL, hInstance );

      CMyWindow wnd;

      wnd.Create( NULL, CWindow::rcDefault, _T("Hello"),
             WS_OVERLAPPEDWINDOW|WS_VISIBLE );
      CMyDialog dlg1;
      dlg1.Create(wnd);

      // If the dialog box resource does not have the WS_VISIBLE style 
invoke
      // ShowWindow

      dlg1.ShowWindow( SW_SHOW );
}
```

The following code example uses the BEGIN_MSG_MAP message map macro. Message map associates a handler function with a particular message, command, or notification. By using message map macros of ATL, you can specify a message map for a window. The window procedures in **CWindowImpl**, **CDialogImpl**, and **CContainedWindow** direct messages form a window to its message map.

Note Additional information on message maps is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcmfc98/html/_atl_message_maps.asp.

```
class CMyDialog: public CDialogImpl<CMyDialog>
{
public:
  enum { IDD = IDD_DIALOG1 };
  BEGIN_MSG_MAP( CMyDialog )
   MESSAGE_HANDLER( WM_INITDIALOG, OnInitDialog )
       MESSAGE_HANDLER( WM_CLOSE, OnClose )
  END_MSG_MAP()
  ……
```

Creating Controls

Controls, such as X Windows widgets, come in all shapes, sizes, colors, and functions. There are two ways to create controls in a Windows API environment. The first and simplest method is by using one of the many dialog box editors. Use these tools to drag and drop controls onto a window or dialog box, which in X Windows is a widget itself. In the Windows API, controls are also windows in every respect. The second method is to call the **CreateWindow** function with the necessary parameters to produce the desired control at the desired location inside a parent window. This section describes the usage of X Windows widgets and Windows controls. An X Windows client can create a control or widget, as shown in the following example.

X Windows example: Display a control on parent window

```
main (int argc, char *argv[] )
{
Widget toplevel; /* Conceptual Application Window */
Widget button;
XtAppContext app; /* context of the app */
toplevel = XtVaAppInitialize( &app, "Example",
NULL,0,&argc,argv,NULL,NULL );
button = XtVaCreateManagedWidget( "command", // button text
commandWidgetClass, //the type of widget
toplevel, // parent widow or parent widget
XtNheight, 50,
XtNwidth, 100,
XtNlabel, "Press To Exit",
NULL );
}
```

(Source File: U_CreateCtrl-UAMV3C6.01.c)

Following is a Windows example to display the control on a parent window.

Windows example: Display a control on parent window

```
HWND handleToThisButton;
handleToThisButton =
CreateWindow( "BUTTON", // the type of control
"Fire Phasers",// button text
WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,// the button style
XpositionInParent,
yPositionInnParent,
BUTTONWIDTH,
BUTTONHEIGHT,
handleOfParentWindow,// parent window
(HMENU)NUMBER_USED_TO_ID_THIS_CONTROL,
hInst,
NULL );
```

Identifying a Control

To communicate with or respond to a control, it is necessary to identify it. This is done using the window handle and a unique ID associated with the control. You can use the handle to a control just as a handle to any window. For example, calling **SetWindowPos** with the window handle of the control can move the control or make it larger. Use the ID in the **WindProc** switch statement to send or capture messages to and from the control.

The following API calls use the ID of the control along with the handle of the parent window:

- **SetDlgItemText**
- **GetDlgItemText**
- **GetDlgItemInt**
- **SetDlgItemInt**

If the developer uses **CreateWindow** to build the control, both identification pieces—the ID of the control and the handle of the parent window—are known. **CreateWindow** returns the handle to the control, and the ninth parameter in the call to **CreateWindow** is the unique ID the application associates with that control.

Communicating with a Control

After the application identifies a control, it can communicate with it. The following are some examples of sending and receiving commands or messages from controls.

X Windows example: Sending and receiving commands or messages from control

```
// X11/Motif
//
XmString newString;
newString = XmStringCreateLocalized("String One");
XmListAddItem( listWidget, newString, 0);
XmStringFree(newString);
newString = XmStringCreateLocalized("String Two");
XmListAddItem( listWidget, newString, 0);
XmStringFree(newString);
newString = XmStringCreateLocalized("String Three");
```

```
XmListAddItem( listWidget, newString, 0);
XmStringFree(newString);
```

Windows example: Sending and receiving commands or messages from control

```
//+
// programmatically add strings to the list box , part of the Dialog
Procedure is // given below
//-
// ATL
//
class CMyDialog: public CDialogImpl<CMyDialog>
{
public:
  enum { IDD = IDD_DIALOG1 };
  BEGIN_MSG_MAP( CMyDialog )
        COMMAND_ID_HANDLER( IDOK, OnOK )
        COMMAND_ID_HANDLER( IDCANCEL, OnCancel)
   MESSAGE_HANDLER( WM_INITDIALOG, OnInitDialog )
       MESSAGE_HANDLER( WM_CLOSE, OnClose )
  END_MSG_MAP()

  LRESULT OnInitDialog( UINT, WPARAM, LPARAM, BOOL& )
  {
      // Get the handle to the ListBox control whose ID is IDC_LIST1
    hList = GetDlgItem(IDC_LIST1 );
      i_count = 0;
   return 0;
  }

  LRESULT OnClose( UINT, WPARAM, LPARAM, BOOL& )
      {

        DestroyWindow();
        return 0;
      }
  LRESULT OnOK( UINT, WPARAM, HWND, BOOL& )
      {
            // Get the text from the Edit control whose ID is IDC_EDIT1
             GetDlgItemText( IDC_EDIT1 , str,20 );
            // Add items to the list box on clicking OK
             SendMessage(hList,LB_ADDSTRING,i_count,(LPARAM) str);
             i_count++;
             return 0;
      }
  LRESULT OnCancel( UINT, WPARAM, HWND, BOOL& )
      {
       //Close the Dialog on clicking Cancel
        DestroyWindow();
        return 0;
      }



 TCHAR str[22];
 HWND hList;
 int i_count;
};
```

Note **SendMessage** returns a LRESULT and takes as parameters HWND, an UINT, a WPARAM, and an LPARAM.

The WPARAM and LPARAM data types have been tailored for the appropriate operating systems such that they are 32 bits on a 32-bit target operating system and 64 bits on a 64-bit platform.

So when passing an LPARAM-type value as the fourth parameter, the cast, if any, should be done to LPARAM and not to DWORD so that the code is 64-bit compliant.

In the case of ATL, we can set the focus to a particular control on the dialog by first attaching the child control to a Window handle and then invoking **SetFocus**, as shown in the code of an event handler.

```
// Windows ATL
LRESULT OnFocusListBox( UINT, WPARAM, HWND, BOOL& )
      {
        CWindow hWnd;
            hWnd.Attach(hList);
            hWnd.SetFocus();
          return 0;
      }
```

In the case of a modeless dialog box, to ensure that the TAB key and mnemonics work properly, a call to **IsDialogMessage** must be made in the main message loop of the application as shown. For more information on **IsDialogMessage**, refer to MSDN®.

```
while( GetMessage( &msg, NULL, 0, 0 ) )
{
        if ( dlg1 ==NULL || !IsDialogMessage(dlg1, &msg) )
        {
            TranslateMessage( &msg );
            DispatchMessage( &msg );
        }
}
```

Table 6.1 describes X Windows and Windows routines for setting focus and selecting the list box item.

Table 6.1. X Windows and Windows Routines

X Windows	Windows
XSetInputFocus(Display display, Window focus, int RevertToParent, Time timeNow)	HWND **SetFocus**(HWND hwnd) .
XmListSelectPos(Widget listWidget, int position, Boolean notify)	LRESULT **SendMessage**(HWND hwnd, UINT message,WPARAM wParam,LPARAM lParam)

Libraries and Include Files

Despite differences in their underlying architectures, many of the graphical functions used in X Windows and Windows perform similar tasks. These include the core libraries and the Motif and Windows API common dialog boxes. You can identify the necessary header files and libraries to port UNIX/Motif core GUI components and common dialogs to the Windows UI.

Core Libraries

A number of functions exist to support the core API used in a graphical user interface. X Windows includes the libraries X and Xlib and the X Windows Intrinsics toolkit. The Windows API equivalent is Windows.h, which includes a large number of additional header files.

The Microsoft compiler includes Windows API USER32.LIB and GDI32.LIB import libraries, which can be roughly compared to X Library and X Toolkit Intrinsics because they provide nearly all of the basic window management and 2-D graphics APIs. These Windows API libraries are called import libraries because they provide information to the linker. When a Windows API-based program references the **CreateWindow** function, User32.lib tells the linker that this function is in the User32.dll dynamic-link library. This information goes to the .exe file, which enables Windows to perform dynamic linking by using the User32.dll and Gdi32.dll DLLs when the program is executed.

Motif and Windows API Common Dialog Boxes

Dialog box functionality is provided by Motif and by the Windows Common Dialog Box Library. If the code migrates from Motif, there is probably an equivalent Windows API common dialog box for each Motif function.

Windows API provides a set of functions to create commonly used windows. If Commdlg.h is included in a project, the project has access to the Windows API common dialog box functions. The Comdlg32.dll library stores templates for these dialog boxes, along with the code to drive them. By using these, a developer can save time and provide a consistent look and feel in the application being migrated.

For example, the Motif function **XmCreateFileSelectionDialog** is very similar to the Windows API function **GetOpenFileName**. The X/Motif code must include the Xm/ FileSB.h header file. The Windows API-based application must include Commdlg.h and must link to Comdlg32.lib. Calling the **GetOpenFileName** API displays the **Open File** dialog box.

In addition to Windows.h, the Windowsx.h library has roots in Windows version 3.1 and provides many useful macros that can be used in code migration. These macros are not used as often or in the same ways now, but they can be helpful. For example, using the **SelectPen** and **DeletePen** macros can be more intuitive than calling **SelectObject** and **DeleteObject** with all the required type specifications:

```
#define SelectPen(hdc, hpen)((HPEN)SelectObject((hdc),
(HGDIOBJ)(HPEN)(hpen)))
#define DeletePen(hpen) DeleteObject((HGDIOBJ)(HPEN)(hpen))
```

Event Handling

This section describes the X Windows and Windows mouse and key board event handling messages. You can use this information to identify the suitable windows routines for the existing X Windows mouse and keyboard event handling routines.

Capturing Mouse Events

There are more than 30 mouse input messages, divided into two cases:

- Client-area mouse messages
- Nonclient-area mouse messages

A window receives client-area mouse messages when a mouse event occurs in the client area of that window. The file Winuser.h (included by Windows.h) defines these message values, as listed in Table 6.2.

Table 6.2. Mouse Event Definitions

Message	Event
WM_LBUTTONDBLCLK	The left mouse button was double-clicked.
WM_LBUTTONDOWN	The left mouse button was pressed.
WM_LBUTTONUP	The left mouse button was released.
WM_MBUTTONDBLCLK	The middle mouse button was double-clicked.
WM_MBUTTONDOWN	The middle mouse button was pressed.
WM_MBUTTONUP	The middle mouse button was released.
WM_RBUTTONDBLCLK	The right mouse button was double-clicked.
WM_RBUTTONDOWN	The right mouse button was pressed.
WM_RBUTTONUP	The right mouse button was released.
WM_XBUTTONDBLCLK	Windows 2000 or Windows XP: An X mouse button was double-clicked.
WM_XBUTTONDOWN	Windows 2000 or Windows XP: An X mouse button was pressed.
WM_XBUTTONUP	Windows 2000 or Windows XP: An X mouse button was released.

Client and Nonclient-Area Mouse Messages

A window receives client-area mouse messages when a mouse event occurs within the client area and nonclient-area mouse messages when a mouse event occurs in the window outside the client area. The nonclient area includes the border, title bar, scroll bar, menu, and minimize and maximize buttons. Each client area message has a corresponding nonclient-area message. A nonclient-area message is defined by including NC in its name, for example, WM_NCLBUTTONUP.

The lParam member of the MSG structure consists of two SHORT values representing the POINTS structure shown in the following code. This can give the current location of the mouse pointer. For client-area messages, the (x,y) pair is relative to the client area of the window. For nonclient-area messages, the (x,y) pair is relative to the upper-left corner of the screen.

The following code shows the handler for the message WM_LBUTTONDOWN that represents client-area mouse messages. The handler determines the (x,y) coordinates of the mouse pointer. (This example shows only the code relevant to the handler.)

Windows example: Handling mouse messages

```
BEGIN_MSG_MAP( CMyDialog )
MESSAGE_HANDLER(WM_LBUTTONDOWN, OnLButtonDown)
MESSAGE_HANDLER(WM_NCLBUTTONUP, OnNCLButtonUP)
END_MSG_MAP()

LRESULT OnLButtonDown(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bVal)
      {
            POINT pnt;
            pnt.x = LOWORD(lParam);
            pnt.y = HIWORD(lParam);
            return 0;
      }
LRESULT OnNCLButtonUP(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bVal)
      {
            MessageBox("Non Client Area : Mouse Button UP");
            return 0;
      }
```

The process for handling client-area mouse messages in X11 is very similar, as shown in the following code.

X Windows example: Handling mouse messages

```
void main() {
Display *xdisplay;
XEvent xEvent;
int mouseX;
int mousey;
while (1) {

/* wait for next event */
XNextEvent (xdisplay, &xevent);
switch (xevent.type) {
case ButtonPress:
mouseX = xevent.xbutton.x;
mousey = xevent.xbutton.y;
break;
}
}
```

Capturing Keyboard Events

In UNIX, a KeyPress event is generated when the user presses a key on the keyboard. Similarly, a KeyRelease event is generated when the user releases the pressed key. Both events arrive with the XKeyEvent structure. A KeySym is a portable number to identify a key with a given engraving. The Xlib function, XLookupString, converts a KeyPress event into both a KeySym and an ASCII string. After you have a KeySym, you can deal with the key itself. The Xlib function **XKeysymToString** returns the string name for a given KeySym. The following example illustrates how these functions are used in capturing the keyboard events in UNIX.

X Windows example: Process keyboard events

```
void PrintKeyEvent(XKeyEvent* event)
{
    KeySym keysym;
    XComposeStatus compose_status;
    int length;
    char string[10];
    switch(event->type)
    {
        case KeyPress:
            printf("KeyPress ");break;
        case KeyRelease:
            printf("KeyRelease ");break;
        default:
            printf("not a key event \n");
            return;
    }
    length = XLookupString(event,string,9,&keysym,&compose_status);
    if((length > 0) && (length <=9))
    {
        string[length]='\0';
        printf("result of xlookupstring [%s] ", string);
    }
    printf("keysym [%s] \n", XKeysymToString(keysym));
}
```

In Windows, a scan code identifies each physical key on the keyboard. The device driver responsible for servicing the keyboard maps this number to a virtual-key code. The include file Winuser.h defines these virtual key codes. After mapping the scan code, the system places a message that includes the scan code and virtual key code along with other information in the system message queue. Some additional system processing takes place, and then the system sends the keyboard message to the process that has the keyboard focus.

Pressing a key causes a WM_KEYDOWN or WM_SYSKEYDOWN message to be placed in the thread message queue attached to the window that has the keyboard focus. Releasing a key causes a WM_KEYUP or WM_SYSKEYUP message to be placed in the queue.

The system posts a WM_CHAR message to the window with the keyboard focus when the **TranslateMessage** function translates a WM_KEYDOWN message. The WM_CHAR message contains the character code of the key that was pressed.

The following example shows how to manually catch and process keystrokes.

Windows example: Process keyboard events

```
//+
// contrived union used only to show how the bits of the
// lParam parameter are arranged
// when handling WM_KEYDOWN messages
//-
typedef union {
struct {
unsigned long repeatCount :16;
unsigned long scanCode :8;
unsigned long extendedChar :1;
unsigned long reserved :4;
unsigned long altKeyDown :1;
unsigned long previousState :1;
unsigned long transition :1;
}bits;
LPARAM lParam;
}tyKeyData;

BEGIN_MSG_MAP( CMyDialog )
MESSAGE_HANDLER(WM_KEYDOWN, OnKeyDown)
MESSAGE_HANDLER(WM_CHAR, OnChar)
END_MSG_MAP()
```

```
//+
// ATL WM_KEYDOWN event handler
//-
LRESULT OnKeyDown(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled)
{
tyKeyData keyData;
TCHAR characterCode;

//+
// just for clarity showing what is in
// the wParam parameter when WM_KEYDOWN is
// sent to the window proc
//-
characterCode = ((TCHAR)(wParam));
//+
// the tyKeyData union is defined above
// this union displays how the bits are defined
//-
keyData.lParam = lParam;
if ( keyData.bits.altKeyDown ) {
//+
// using the keyboard hardware scan code
// to determine what key was pressed
//-
switch ( keyData.bits.scanCode ) {
case 0x3B : // <Alt-F1>
break;
case 0x3C : // <Alt-F2>
break;
default :
break;
}
}
else {
//+
// VK_XX Key Codes are found in winuser.h
// These are not the keyboard hardware scan codes!!!
// using the wParam to determine
// what key was pressed
//-
switch ( characterCode ) {
case VK_F1 : // <F1>
break;
case VK_F2 : // <F2>
break;
default :
break;
}
}
return 0;
}

LRESULT OnChar(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
{
  characterCode = ((TCHAR)(wParam));
switch ( characterCode ) {
//+
// VK_XX codes can be used here
// VK_XX Key Codes are found in winuser.h
//-
case 0x08: // backspace
case 0x0A: // linefeed
case 0x1B: // escape
break;
case VK_LEFT : // left arrow
case VK_UP : // up arrow
case VK_INSERT : // the insert key
break;
//+
// convert TAB to Spaces
//-
case 0x09: // tab
for ( int i = 0; i < 4; i++)
SendMessage(hWnd, WM_CHAR, 0x20, 0);
}
return 0;
}
```

Keyboard Focus

Keyboard focus is a temporary property of a window or widget. At any given time, only one component can listen to the keyboard for events. The window or widget that is listening is said to have the current focus, keyboard focus, or just focus. Processing focus in a Windows API-based application involves processing the WM_KILLFOCUS and WM_SETFOCUS Windows messages. This is similar to using **XmNfocusCallback** and **XmNlosingFocusCallback** for focus callbacks set up within X/Motif.

The following code shows the window procedure for a subclassed button that is handling focus messages:

```
LRESULT CALLBACK CSoftKeyProc(HWND hWnd,UINT iMsg, WPARAM wParam,LPARAM lParam)
{
LRESULT lResult = FALSE;
//+
// this is a trick to retrieve data (in this case a pointer ) that is attached to // this window object.
// SetWindowLongPtr() was used to initially attach this data to the window.
//The reason for this
// being used here is that this function may be the callback for any number
//of these type
// objects and this data is "state" for this particular instance
//-
CvtSoftKey *pSoftKey = (CvtSoftKey *)GetWindowLongPtr( hWnd, 0 );
switch (iMsg) {
default :
break;
//+
// this window (button in this case) is receiving the focus
// so we can do whatever processing we like
// Draw a new border – Highlight the text – whatever!
//-
case WM_SETFOCUS :
lResult = pSoftKey->OnSetFocus( hWnd,iMsg,wParam,lParam);
break;
//+
// this window ( button in this case ) is losing focus
//-
case WM_KILLFOCUS :
lResult = pSoftKey->OnKillFocus( hWnd,iMsg,wParam,lParam);
break;
}
return DefWindowProc(hWnd,iMsg,wParam,lParam);
}
```

Note If a pointer or a handle is being retrieved, the **GetWindowLongPtr** function supersedes the **GetWindowLong** function. (Pointers and handles are 32 bits on 32-bit Windows and 64 bits on 64-bit Windows.) To write code that is compatible with both 32-bit and 64-bit versions of Windows, use **GetWindowLongPtr**.

Creating Keystrokes, Mouse Motions, and Button Click

You can simulate keystrokes, mouse motions, or button clicks by using the **SendInput** function to serially insert events into the mouse or keyboard stream.

Note Additional information about handling the keyboard is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/userinput/keyboardinput/keyboardinputreference/keyboardinputfunctions/keybd_event.asp.

Graphics Device Interface

The Graphics Device Interface (GDI) is a set of API functions and data structures that can be used to generate graphics for devices such as displays and printers. These functions help you in creating such graphic objects as Pens, Brush, and Palette to draw such shapes as lines, circles, and rectangles. Functions are available to display the text of different fonts and to draw images as well.

This section describes the GDI-specific routines and functions used in X Windows applications and their corresponding replacements in the Windows environment. You can use this information to identify the best approach to migrate the GDI-specific routines in your UNIX application to the Windows environment using the Win32/Win64 API.

Device Context

Applications on both platforms use a context to control how drawing functions behave. On X Windows systems, this context is known as the graphics context (GC). On Windows API-based GDI systems, this context is known as the device context (DC). One difference between the two platforms is in the location where the operating system stores and manages drawing attributes such as the width of lines or the current font.

In X Windows, these values belong to the graphics context. When using **XCreateGC** or **XtGetGC**, it is necessary to provide a values mask and values structure. These values are used to store settings such as line width, foreground color, background color, and font style.

The following code is an example of the process of setting the foreground and background colors:

```
GC gcRedBlue;
XGCValues gcValues;
unsigned long gcColorRed;
unsigned long gcColorBlue;
unsigned long gcColorWhite;
Widget myWidget;
int main (int args, char **argv)
{
// initialize colors - widget - etc.
gcValues.foreground = gcColorRed;
gcValues.background = gcColorBlue;
gcRedBlue = XtGetGC ( myWidget, GCForeground | GCBackground,
&gcValues);
}
```

Windows API-based applications use a different approach. The device context is a structure that defines a set of graphic objects and their associated attributes and the graphic modes that affect output. To keep this simple, the graphic objects include a font for displaying text, a pen for line drawing, and a brush for painting and filling. To draw lines, rectangles, and text, it is necessary to get or create one of these objects and select it into the desired device context.

Instead of creating several specialized graphics context objects as X Windows does, Windows API-based applications create several drawing objects and then select them into the device context as and when required. This methodology is similar to what an X Windows client application could do by getting a single graphics context and then repeatedly calling **XChangeGC**. The following code shows a Windows API-based application that creates several pens and uses them to draw lines and rectangles; the same code could be called from an event handler member function in the case of ATL classes:

```
#define onePixel 1
#define threePixels 3
#define thinLine onePixel
#define thickLine threePixels
COLORREF colorRed;
COLORREF colorBlue;
void drawSomthing( HDC hDC )
{
HPEN thinRedPen;
HPEN thinBluePen;
HPEN oldPen;
int x;
int y;
// initialize colors - etc.
//+
// create two pens.
// this could be done more statically somewhere so that
// it would not be necessary to create them each time
// this method is called.
//-
colorRed = RGB(255,0,0);
colorBlue = RGB(0,0,255);
thinRedPen = CreatePen( PS_SOLID, thinLine, colorRed );
thinBluePen = CreatePen( PS_SOLID, thinLine, colorBlue );
x = 100;
y = 200;
//+
// draw a line with the current pen,
// whatever it is at this time for this DC
//-
LineTo( hDC, x,y );
//+
// make our pen the current pen for the DC
// and save the existing one so we can put it back
//-
oldPen = (HPEN)SelectObject( hDC, thinRedPen );
//+
// draw a line with our pen
//-
LineTo( hDC, x,y );
//+
// make our other pen current in the DC.
// we are not saving the old one.
//-
SelectObject ( hDC, thinBluePen );
//+
// draw a line using our second pen
//-
LineTo( hDC, x, y );
//+
// put back the original pen
//-
SelectObject( hDC, oldPen );
//+
// get rid of our pen resources
//-
DeleteObject( thinRedPen );
DeleteObject( thinBluePen );
}
```

Getting Windows GDI Device Context

Windows API-based applications can retrieve the device context from the window handle, as shown in the following example:

```
void myFunction ( HWND hWnd )
{
//+
// The following example attaches the Window handle hWnd to the CWindow
object
// and calls CWindow::GetDC to retrieve the DC of the client
// area of the window wrapped by CWindow Object.
//-
CWindow myWindow;
myWindow.Attach(hWnd);
HDC hDC = myWindow.GetDC();// draw using the device context hDC
//+
// release the DC
//-
myWindow.ReleaseDC(hDC);
hDC = NULL;
```

Windows API-based applications can also retrieve the device context from the window handle by using **BeginPaint** and **EndPaint**, as shown in the following example:

```
LRESULT CALLBACK WndProc( HWND hWnd,
UINT message,
WPARAM wParam,
LPARAM lParam)
{
HDC hDC;
PAINTSTRUCT ps;
switch ( message ) {
case WM_PAINT:
//+
// Retrieve the device context (DC)
// for the window referenced by hWnd
//-
hDC = BeginPaint( hWnd, &ps );
//+
// draw with hDC or ps.hdc
//-
//+
// always follow BeginPaint() with EndPaint()
//-
EndPaint( hWnd, &ps );
break;
}
}
```

Creating Windows API GDI Device Context

It is often useful to draw in an off-screen buffer and then move that buffer into the display memory. This hides the live drawing function calls from the user and eliminates flicker in the window. This technique is called double buffering and is used extensively for programming games.

To create this off-screen context

1. Calculate the width and height that are needed.
2. Get the device context of the target (dialog box, button, or any other window object).
3. Call CreateCompatibleDC.
4. Call CreateCompatibleBitmap.

```
// m_hButton is the window handle to a button.
// m_clientRect is a RECT structure.
// Step 1. Calculate the size.
GetClientRect( m_hButton, &m_clientRect );
m_width = ((int)( m_clientRect.right - m_clientRect.left ));
m_height = ((int)( m_clientRect.bottom - m_clientRect.top ));
// Step 2. Get the DC of the target window.
hdc = GetDC( m_hButton );
// Step 3. Create a compatible device context.
m_hdcMem = CreateCompatibleDC(hdc);
// Step 4. Create a compatible bitmap - our X Windows drawable.
m_hbmpMem = CreateCompatibleBitmap( hdc,m_width,m_height );
```

To use and display this off-screen bitmap

1. Select the compatible bitmap into the compatible device context.
2. Draw on that device context.
3. Get the target window device context.
4. Transfer the compatible memory image to the screen.
5. Select the old bitmap into the device context.

```
// Step 1. Select the compatible bitmap into the compatible DC.
// hbmpOld is a handle to a bitmap
// m_hdcMem is the compatible device context
// m_hbmpMem is the compatible bitmap
hbmpOld = (HBITMAP)SelectObject( m_hdcMem, m_hbmpMem );
// Step 2. Draw on that DC.
// FillRect() cleans out the rectangle
FillRect( m_hdcMem, &m_clientRect, hBackgroundBrush );
// Draw a line
LineTo( m_hdcMem, x,y );
// Step 3. Get the target DC.
targetDC = GetDC( hTargetWindow );
// Step 4. Transfer the compatible image to the screen.
// transfer everything to the screen
// hdcMem is what we drew on
//-
BitBlt( targetDC,
0,
0,
m_width,
m_height,
m_hdcMem,
0,
0,
SRCCOPY );
// Step 5. Put the old bitmap back into the compatible DC.
```

```
SelectObject( m_hdcMem, hbmpOld );
// based on program logic - Release the DC of the target window
ReleaseDC( hTargetWindow, targetDC );
```

For more information about Windows API-based GDI device context, search the MSDN Web site for **GetDC**, **CreateDC**, **CreateCompatibleDC**, and **DeleteDC**.

Display and Color Management

The X Windows and Windows API-based GDI are both constrained by the physical limitations of the available display hardware. One such limitation is the number of colors a display adapter is capable of showing. You can use the information provided in this section to identify the corresponding Windows routines for Drawing and Graphics Device management. You will also be able to port X Windows applications with drawing and graphics code to the Windows environment.

All X Windows applications use a color map. This map can be shared or private. A shared color map is used by all other applications that do not use a private map. Using a private map gives an application better color control and potentially a greater number of colors. There is one problem with private maps: When the mouse moves on or off the client that is using a private map, the screen colors change. Windows API-based applications typically use color with no regard for the display device.

If the application uses a color that is beyond the capabilities of the display device, the system approximates that color within the limits of the hardware. On display devices that support a color palette, applications sensitive to color quality can create and manage one or more logical palettes.

A palette is similar to an X Windows color map. Both of these methodologies are used to map some desired colors onto the physical capabilities of the display hardware. For example, if a Windows API-based program needs more than 16 colors and is running on an 8-bits-per-pixel (bpp) display adapter, the program must create and use a palette.

The Windows API system palette can be thought of as being similar to an X Windows shared color map. A logical palette created and realized by an application can be thought of as an X Windows private color map.

A Windows API-based application that uses a logical palette exhibits some of the same behaviors as an X Windows application that uses a private color map. The application that gets priority in color selection is the one with the current focus. When the application that has the current focus calls **RealizePalette**, the system palette changes and the WM_PALETTECHANGED message is sent to all top-level and overlapped windows. This message enables a window that uses a color palette but does not have the keyboard focus to realize its logical palette and update its client area. The wParam parameter identifies the owner window. Inspecting this value prevents the originating window from realizing a logical palette repeatedly upon receipt of this message.

Today, most display hardware is capable of 24-bit or better color depth. For palette examples, see the many samples both on the MSDN Web site and in the Microsoft Windows Platform SDK.

To create a logical color palette

1. Allocate a LOGPALETTE structure and assign values to it.
2. Call CreatePalette with a pointer to the LOGPALETTE structure.
3. Call SelectPalette by using the pointer returned from CreatePalette.
4. Call RealizePalette to make the system palette the same as the device context.
5. Call UnrealizeObject when finished with the palette.

To determine the capabilities of the hardware and to calculate the best possible behaviors of the display, an X Windows program can use such functions as:

- **DefaultColorMap**
- **DefaultVisual**
- **DisplayCells**
- **DisplayPlanes**
- **XGetVisualInfo**
- **XGetWindowAttributes**

A Windows API-based application can rely on **GetDeviceCaps** for this information. The **GetDeviceCaps** function retrieves device-specific information for the specified device. The following code example shows a few examples of device information that can be retrieved by using **GetDeviceCaps**. For a full list of the possible values of the nIndex parameter, refer to the operating system Help or search the MSDN Web site.

```
int GetDeviceCaps(
HDC hdc, // handle to DC
int nIndex // index of capability
);
void myFunction( HWND hThisWindow )
{
HDC hDC;
hDC = GetDC( hThisWindow );
widthOfScreenInPixels = GetDeviceCaps( hDC, HORZRES );
numberOfColorPlanes = GetDeviceCaps( hDC, PLANES );
numberOfColors = GetDeviceCaps( hDC, NUMCOLORS );
numberOfFonts = GetDeviceCaps( hDC, NUMFONTS );
}
```

Drawing 2-D Lines and Shapes

The device context of a drawing surface contains attributes that directly affect how lines, curves, and rectangles are drawn. These attributes include the current brush and pen and the current position.

The default current position for any given device context is (0,0) in logical two-dimensional (2-D) space. The value of the current position can be changed by calling **MoveToEx**, as shown in the following code example. The **MoveToEx** function updates the current position to the specified point and optionally returns the previous position. This function affects all drawing functions.

```
BOOL MoveToEx(
HDC hdc, // handle to device context
int X, // x-coordinate of new current position
int Y, // y-coordinate of new current position
LPPOINT lpPoint // old current position
);
```

The POINT structure defines the x and y coordinates of a point, as shown in the following code example:

```
typedef struct tagPOINT {
LONG x;
LONG y;
} POINT, *PPOINT;
```

Drawing Lines

Two sets of line and curve drawing functions are provided in the Windows API and GDI API. These two sets of functions are identified by the letters "To" at the end of the function name. Functions ending with "To" use and set the current position. Those functions that do not end with "To" leave the current position as it was. The **LineTo** function draws a line from the current position up to, but not including, the specified point, as shown in the following code example:

```
BOOL LineTo(
HDC hdc, // device context handle
int nXEnd, // x-coordinate of ending point
int nYEnd // y-coordinate of ending point
);
```

The **PolylineTo** function draws one or more straight lines that use and update the current position. A line is drawn from the current position to the first point specified by the lppt parameter by using the current pen. For each additional line, the function draws from the ending point of the previous line to the next point specified by lppt, as shown in the following code example:

```
BOOL PolylineTo(
HDC hdc, // handle to device context
CONST POINT *lppt, // array of points
DWORD cCount // number of points in array
);
```

The **Polyline** function draws a series of line segments by connecting the points in the specified array, as shown in the following code example. The lines are drawn from the first point through subsequent points by using the current pen. Unlike the **LineTo** or **PolylineTo** functions, the **Polyline** function neither uses nor updates the current position.

```
BOOL Polyline(
HDC hdc, // handle to device context
CONST POINT *lppt, // array of endpoints
int cPoints // number of points in array
);
```

X Windows example: Drawing lines

The following X Windows example shows the use of **XDrawLine**.

```
int main (int argc, char **argv)
{
XtToolkitInitialize ();
myApplication = XtCreateApplicationContext ();
myDisplay = XtOpenDisplay( myApplication,
NULL,
NULL,
"XBlaat",
NULL,
0,
&argc,
argv);
myWindow = RootWindowOfScreen(DefaultScreenOfDisplay (mydisplay));
//+
// now we need a surface to draw on
//-
myMap = XCreatePixmap ( myDisplay,myWindow,64,64, 1 );
values.foreground =
BlackPixel (myDisplay, DefaultScreen (myDisplay));
myGC = XCreateGC (myDisplay, mySurface, GCForeground, &values);
//+
// draw two diagonal lines across the 64x64 surface
//
XDrawLine( myDisplay,mySurface,myGC,0,0,63,63 );
XDrawLine( myDisplay,mySurface,myGC,0,63,63,0 );
…
}
```

Windows example: Drawing lines

The following Windows ATL example shows the use of **MoveToEx** and **LineTo.**

```
BEGIN_MSG_MAP( CMyWindow )
       MESSAGE_HANDLER(WM_LBUTTONDOWN, OnLButtonDown)
       MESSAGE_HANDLER(WM_LBUTTONUP, OnLButtonUP)
       MESSAGE_HANDLER(WM_MOUSEMOVE, OnMouseMove)
END_MSG_MAP()
POINTS current;
POINTS start;
LRESULT OnMouseMove(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled)
{

COLORREF red = RGB(255,0,0);
// retrieve the current position from lParam
current.x = LOWORD(lParam);
current.y = HIWORD(lParam);

//Obtain the current device context
HDC hdc = GetDC();

//Create the red pen
HPEN pen = CreatePen(PS_SOLID, 2, red);

//Save the oldpen
HPEN oldpen = (HPEN)SelectObject(hdc, pen);

if (start.x != -1 )
{
//Move to the starting point selected in MouseUp event
       MoveToEx(hdc, start.x, start.y, NULL);
//Draw a line from the previous point to the current point
       LineTo(hdc, current.x, current.y);
       start.x = current.x;
       start.y = current.y;
}
}
```

Drawing Rectangles

In the Windows API, a rectangle shape is a filled shape. Filled shapes are geometric forms that the current pen can outline and the current brush can fill.

Following are the five filled shapes:

- Ellipse
- Chord
- Pie
- Polygon
- Rectangle

In X Windows, the XRectangle shape is quite different from the Windows API equivalent. When porting between the two, it is necessary to understand the conceptual difference.

The X Windows version uses an upper-left–corner point and the width and height. The Windows API version uses the upper-left and lower-right points. This difference is also true for the **XDrawRectangle** and Windows API **Rectangle** functions.

The X Windows structure is as follows:

```
typedef struct {
short x,y;
unsigned short width,height;
} XRectangle;
```

Its Windows API equivalent is as follows:

```
typedef struct _RECT {
LONG left;
LONG top;
LONG right;
LONG bottom;
} RECT, *PRECT;
```

The **Rectangle** function draws a rectangle and is outlined by using the current pen and filled by using the current brush. Because it does not fill the rectangle, this function is quite different from the **XDrawRectangle** function.

```
BOOL Rectangle(
HDC hdc, // handle to DC
int nLeftRect, // x-coord of upper-left corner of rectangle
int nTopRect, // y-coord of upper-left corner of rectangle
int nRightRect, // x-coord of lower-right corner of rectangle
int nBottomRect // y-coord of lower-right corner of rectangle
);
```

Rectangle functions that fill the rectangle are as follows:

- X Windows: **XFillRectangle**
- Windows API: **Rectangle**
- Windows API: **FillRect**

Rectangle functions that draw the outline only are as follows:

- X Windows: **XDrawRectangle**
- Windows API: **FrameRect**

Note The Windows API functions **Rectangle** and **FillRect** differ in the parameters they take. For more information, refer to the Visual Studio® Help or the MSDN Web site.

X Windows example: Handling rectangle functions

The following X Windows example demonstrates rectangle functions:

```
void drawSomeRectangles()
{
//+
// fill the rectangle and then draw a black border around it
//-
XFillRectangle (myDisplay, mySurface, myWhiteGC, 0, 0, 31, 31);
XDrawRectangle (myDisplay, mySurface, myBlackGC, 0, 0, 31, 31);
//+
// draw an empty rectangle ten pixels square
//-
XDrawRectangle( myDislay, mySurface, myBlackGC, 0,0, 10,10 );
}
```

Windows example: Handling rectangle functions

The following ATL code example demonstrates rectangle functions on the Mouse left button down event:

```
LRESULT OnLButtonDown(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled){
RECT myRectangle;
//+
// fill the rectangle and then draw a black border around it
// assume that the current pen in this DC is black and the
// current brush selected is red
// The rectangle top left corner is the point where the left mouse
button
// is clicked
//-
HBRUSH myredBrush = CreateSolidBrush(RGB(255, 0, 0));
SelectObject(hdc, myredBrush);
Rectangle( hDC, LOWORD(lParam), HIWORD(wParam),31,31 );
//+
// draw an empty rectangle ten pixels square
// The FrameRect() function draws a border around the specified
// rectangle by using the specified brush rather than the current
// pen.
//
// The width and height of the border are always one logical unit.
//-
myRectangle.top = 0;
myRectangle.left = 0;
myRectangle.bottom = 10;
myRectangle.right = 10;
HBRUSH myGreenBrush = CreateSolidBrush(RGB(255, 0, 0));
FrameRect( hdc, &myRectangle, myGreenBrush );
}
```

Windows Character Data Types

This section describes various routines and functions related to fonts and character sets used in X Windows applications and their alternatives in the Windows environment. You can use the information provided in this section to identify the front-specific and text-specific routines in your UNIX applications and implement the replacements in the Windows environment using the Win32/Win64 API.

Most of the pointer-type names are prefixed with a P or LP. For more information about character sets used by fonts, see the operating system Help documentation, or search the MSDN Web site. A best practice with characters is to declare all characters and strings as TCHAR and use the TEXT macro to declare static strings. For example:

```
TCHAR myString[255];
wsprintf( myString,
TEXT("This is a good example %d is a %s \n" ),
1950,
TEXT("Year")
);
```

For more information about **wsprintf** and the rest of the string functions, see the operating system Help documentation or search the MSDN Web site.

Text and Fonts

Text can be formatted by creating and using fonts and making intelligent decisions about mapping modes and kerning.

Displaying Text

This section discusses the use of fonts to display text.

Using Fonts

Fonts control the display characteristics of text. An X Windows client application can use the **XLoadQueryFont** and **XSetFont** functions to apply a font to a given graphics context, as shown in the following code.

X Windows example: Applying a font to given graphics context

```
#define FONT1 "-*-lucida-medium-r-*-*-12-*-*-*-*-*-*-*"
Font font1;
XFontStruct *font1Info;
main() {
Display *pDisplay;
int iScreen;
GC gc;
pDisplay = XOpenDisplay("myDisplay");
iScreen = DefaultScreen(pDisplay);
//+
// get the Graphics Context
//-
gc = DefaultGC(pDisplay,iScreen);
//+
// attempt to load the font
//-
font1Info = XLoadQueryFont( pDisplay,FONT1 );
font1 = font1Info->fid;
//+
// Set the font in the GC
//-
XSetFont( pDisplay, gc, font1 );
…
…
```

A Windows API-based application follows the same logic. That is, it creates or selects a font, retrieves a device context, and then selects the font object for that device context. This is shown in the following example.

Windows example: Applying a font to given graphics context

```
#define FONT1 TEXT(“Lucida Console”);
HFONT hFont1;
void fontDemo( HWND hWnd )
{
HDC hDC;
HFONT hOldFont;
//+
// get the Device Context
//-
hDC = GetDC( hWnd );
//+
// attempt to load the system font
//-
hFont1 = (HFONT)GetStockObject (SYSTEM_FONT);
//+
// Set the font in the GC
//-
hOldFont = (HFONT)SelectObject( hDC, hFont1 );
…
```

Creating Fonts

The short example in this section uses the font specified by SYSTEM_FONT, which duplicates the default, although developers are likely to use something more creative. The Windows API **CreateFont**, **CreateFontIndirect**, **CreateScalableFontResource**, and **CreateFontIndirectEx** functions provide the capability to create logical fonts based on the fonts loaded on the system.

Windows example: Creation of fonts

```
#define MY_FONT_FACE TEXT("Lucida Console")
//+
// fontAttribute Option Bits
//-
#define fontAttribute_BOLD 0x01
#define fontAttribute_CROSSED_OUT 0x02
#define fontAttribute_UNDERLINED 0x04
#define fontAttribute_ITALIC 0x08
typedef struct {
unsigned char fontSize;
unsigned char fontStyle;
TCHAR *fontFace;
} tyFONT_ATTRIBUTE;
HFONT createFont( tyFONT_ATTRIBUTE *fontAttributeObject )
{
```

```
HFONT hFont;
LOGFONT lf;
//+
// these are completely arbitrary values for this example code.
// they simply associate a width and height with a
// font size number found in the tyFONT_ATTRIBUTE struct.
//
// For example fontSize == 2 (used to index these two arrays)
// will produce a 12x8 font
//-
int fontHeight[] = {8,8,12,16,16,24,32, 32,48,64,64,96,128,128,192};
int fontWidth [] = {6,8, 8,12,16,16,24, 32,32,48,64,64, 96,128,128};
//+
// pick a font face
//-
lstrcpy( lf.lfFaceName, fontAttributeObject->fontFace );
//+
// protect against running out of the arrays above
// and pick a default behavior of "2"
//-
if ( fontAttributeObject->fontSize > 14 )
fontAttributeObject->fontSize = 2;
if ( fontAttributeObject->fontStyle & fontAttribute_BOLD )
lf.lfWeight = FW_MEDIUM;
else
lf.lfWeight = FW_LIGHT;
lf.lfItalic = (unsigned char)( fontAttributeObject->fontStyle &
fontAttribute_ITALIC );
lf.lfUnderline = (unsigned char)( fontAttributeObject->fontStyle &
fontAttribute_UNDERLINED );
lf.lfStrikeOut = (unsigned char)( fontAttributeObject->fontStyle &
fontAttribute_CROSSED_OUT );
lf.lfEscapement = 0;
lf.lfOrientation = 0;
lf.lfCharSet = ANSI_CHARSET;
lf.lfClipPrecision = CLIP_DEFAULT_PRECIS;
lf.lfQuality = DRAFT_QUALITY;
lf.lfPitchAndFamily = FF_MODERN | FIXED_PITCH;
lf.lfHeight = fontHeight [ fontAttributeObject->fontSize ];
lf.lfWidth = fontWidth [ fontAttributeObject->fontSize ];
hFont = CreateFontIndirect(&lf);
return( hFont );
}
```

```
//+
// example using createFont()
//-
void fontDemo( HWND hWnd )
{
HDC hDC;
HFONT hOldFont;
HFONT hFont1;
tyFONT_ATTRIBUTE fontAttribute;
//+
// get the Device Context
//-
hDC = GetDC( hWnd );
//+
// attempt to create a font
//-
fontAttribute.fontSize = 2;
fontAttribute.fontStyle = (fontAttribute_BOLD | fontAttribute_ITALIC );
lstrcpy( fontAttribute.fontFace, MY_FONT_FACE );
hFont1 = createFont( &fontAttribute );
//+
// Set the font in the GC
//-
hOldFont = (HFONT)SelectObject( hDC, hFont1 );
```

Note Additional information about creating and using logical fonts in a Windows API-based application is available on the MSDN Web site at
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnprogwin/html/ch17-03.asp.

Drawing Text

Because simple is often better, this discussion starts with the X Windows **XDrawString** function and the Windows API **TextOut** function. Both functions require a context to draw on the x and y coordinates, the string, and the string length in characters. The examples in this section draw the string “Hello World” in the current font and colors at the specified coordinates.

It is often desirable to set a particular font or color before writing the text. These examples show how the two systems perform these tasks.

A programmer can code font and text display in X Windows as shown in the following code:

```
font = XLoadQueryFont (display, “fixed”);
XSetFont (display, gc, font->fid);
XSetBackground(display, gc, WhitePixel(display, screen));
XSetForeground(display,
gc, BlackPixel(display, screen));
XDrawString( display, d, gc, x, y, “Hello World”, 11 );
```

X Windows provides explicit definitions of 8-bit and 16-bit character functions with **XDrawString** and **XDrawString16**. Likewise, the Windows API provides **TextOutA** for ASCII (8-bit characters) and **TextOutW** for Wide Char (16-bit UNICODE characters).

The **TextOut** function is actually a macro that resolves correctly to **TextOutA** or **TextOutW** based on the status of the UNICODE definition, as follows:

```
#define UNICODE
#define _UNICODE
TextOut()… // this will result in TextOutW()
#undef UNICODE
#undef _UNICODE
TextOut()… // this will result in TextOutA()
```

One drawback of using **XDrawString** and **TextOut** is that nothing is done about erasing the background. Continually outputting strings to the same x and y coordinates results in a jumble of unreadable text strings, one upon the other. The X Windows library provides the **DrawImageString** function, which calculates a rectangle containing the string and fills it with the background pixel color before drawing the text in the foreground pixel color. The Windows API supports the **ExtTextOut** function to provide this capability. Using the Windows API **ExtTextOut** function requires the bounding rectangle to be calculated and passed into the function. This requires knowledge about the current font and logical display units.

A programmer can code font and text display in the Windows API as follows:

```
#define rgbBlack (COLORREF)RGB( 0x00,0x00,0x00 )
#define rgbWhite (COLORREF)RGB( 0xFF,0xFF,0xFF )
font = (HFONT)GetStockObject(OEM_FIXED_FONT);
oldFont = (HFONT)SelectObject( hdc, font ); // save old font
SetTextColor (hdc, rgbBlack);
SetBkColor (hdc, rgbWhite);
TextOut( hdc, x, y, "Hello World", 11);
```

The preceding Windows API code example uses the COLORREF type, the **RGB** macro, and the **GetStockObject** function in the following sequence:

1. The COLORREF value specifies an RGB color and is defined as follows:

   ```
   typedef DWORD COLORREF;
   typedef DWORD *LPCOLORREF;
   ```

2. The **RGB** macro selects the red, green, blue (RGB) color combination based on the arguments supplied and the color capabilities of the output device, as follows:

   ```
   COLORREF RGB(
   BYTE byRed, // red component of color
   BYTE byGreen, // green component of color
   BYTE byBlue // blue component of color
   );
   ```

3. The **GetStockObject(int objectType)** function retrieves a handle to one of the stock pens, brushes, fonts, or palettes. The return value must be cast to the expected type, as follows:

   ```
   void foo() {
   HFONT hFont;
   HBRUSH hBrush;
   hfont = (HFONT)GetStockObject(DEFAULT_GUI_FONT);
   hBrush = (HBRUSH)GetStockObject(BLACK_BRUSH);
   }
   ```

Following are the other APIs that are used for text output:

- **DrawText**
- **DrawTextEx**
- **PolyTextOut**
- **TabbedTextOut**

The **DrawText** function draws formatted text in the specified rectangle, as shown in the following example. It formats the text according to the specified method, expanding tabs, justifying characters, and breaking lines.

```
int DrawText(
HDC hDC, // handle to DC
LPCTSTR lpString, // text to draw
int nCount, // text length
LPRECT lpRect, // formatting dimensions
UINT uFormat // text-drawing options
);
```

To specify additional formatting options, use the **DrawTextEx** function:

```
int DrawTextEx(
 HDC hdc,           // handle to DC
 LPTSTR lpchText,       // text to draw
 int cchText,         // length of text to draw
 LPRECT lprc,         // rectangle coordinates
 UINT dwDTFormat,       // formatting options
 LPDRAWTEXTPARAMS lpDTParams // more formatting options
);
```

The **PolyTextOut** function draws several strings using the font and text colors currently selected in the specified device context.

```
BOOL PolyTextOut(
 HDC hdc,        // handle to DC
 CONST POLYTEXT *pptxt, // array of strings
 int cStrings      // number of strings in array
);
```

The **TabbedTextOut** function writes a character string at a specified location, expanding tabs to the values specified in an array of tab-stop positions. The text is written in the currently selected font, background color, and text color.

```
LONG TabbedTextOut(
 HDC hDC,            // handle to DC
 int X,              // x-coord of start
 int Y,              // y-coord of start
 LPCTSTR lpString,        // character string
 int nCount,           // number of characters
 int nTabPositions,        // number of tabs in array
 CONST LPINT lpnTabStopPositions, // array of tab positions
 int nTabOrigin          // start of tab expansion
);
```

Formatting Text

The formatting functions can be divided into the following three categories:

- Those that retrieve or set the text-formatting attributes for a device context.
- Those that retrieve character widths.
- Those that retrieve string widths and heights.

Text-Formatting Attribute APIs

Text-formatting attribute APIs are a set of APIs used to set or retrieve text-formatting attributes for a device context. The text formatting attributes could be the text alignment, inter character spacing, text justification, or text and background colors.

Table 6.3 lists the APIs and their functions.

Table 6.3. APIs and Their Functions

APIs	Description
SetBkColor	Sets the current background color to the specified color value or to the nearest physical color if the device cannot represent the specified color value.
SetBkMode	Sets the background mix mode of the specified device context.
SetTextAlign	Sets the text alignment flags for a device context.
SetTextCharacterExtra	Sets the intercharacter spacing.
SetTextColor	Sets the text color for a device context.
SetTextJustification	Specifies the amount of space the system should add to the break characters in a string.
GetBkColor	Returns the current background color for the specified device context.
GetBkMode	Returns the current background mix mode for a specified device context.
GetTextAlign	Gets the text-alignment setting for a device context.
GetTextCharacterExtra	Gets the current intercharacter spacing for a device context.
GetTextColor	Gets the text color for a device context.

APIs to Retrieve Character Widths

Applications must retrieve character-width data when they perform such tasks as fitting strings of text to page or column widths.

An application can use the **GetCharWidth32** and **GetCharWidthFloat** functions to retrieve the advance width for individual characters or symbols in a string of text. The advance width is the distance that the cursor on a screen or the print-head on a printer must advance before printing the next character in a string of text. The **GetCharWidth32** function returns the advance width as an integer value. If greater precision is required, an application can use the **GetCharWidthFloat** function to retrieve fractional advance-width values.

An application can retrieve actual character-width data by using the **GetCharABCWidths** and **GetCharABCWidthsFloat** functions. The **GetCharABCWidthsFloat** function works with all fonts. The **GetCharABCWidths** function only works with TrueType and OpenType fonts.

APIs to Retrieve String Widths and Heights

In X Windows you can rely on **XTextWidth** to get the length of a character string in pixels. With the Windows API, you must work a little harder to get this number.

It is necessary to understand the mapping mode. The mapping mode defines the unit of measure used to transform page-space units into device-space units. It also defines the orientation of the x and y axes of the device.

A mapping mode is a scaling transformation that specifies the size of the units used to draw operations. The mapping mode can also perform translation. In some cases, the mapping mode alters the orientation of the x and y axes in the device space.

The default mapping mode is MM_TEXT. One logical unit equals one pixel. Positive x-axis is to the right, and positive y-axis is down. This mode maps directly to the coordinate system of the device.

The Windows API **SetMapMode** function sets the mapping mode of the specified device context, as shown in the following code:

```
int SetMapMode(
HDC hdc, // handle to device context
int fnMapMode // new mapping mode
);
```

To calculate the size of a string in pixels, it is necessary for the current mapping mode to be MM_TEXT. The Windows API programmer can either assume the current mapping mode as the default MM_TEXT and set it to MM_TEXT by calling **SetMapMode**, or make sure it is MM_TEXT by using **GetMapMode** to retrieve it. (For more information, search for MM_TEXT on the MSDN Web site.)

The Windows API **GetTextExtentPoint32** function returns the width and height of a string of text in logical units, as shown in the following code. Recall that setting the mapping mode to MM_TEXT returns logical units as pixels.

```
BOOL GetTextExtentPoint32(
HDC hdc, // handle to DC
LPCTSTR lpString, // text string
int cbString, // characters in string
LPSIZE lpSize // string size
);
```

The size structure resembles the following code, and is defined in Windef.h:

```
typedef struct tagSIZE {
LONG cx;
LONG cy;
} SIZE, *PSIZE, *LPSIZE;
```

The Windows API **GetTextMetrics** function fills a TEXTMETRIC structure with all the information about the currently selected font of the device context, as shown in the following code. The programmer can use this information to perform any number of scaling or text size calculations.

```
BOOL GetTextMetrics(
HDC hdc, // handle to DC
LPTEXTMETRIC lptm // text metrics
);
```

The TEXTMETRIC structure contains basic information about a physical font, as shown in the following example. All sizes are specified in logical units; that is, they depend on the current mapping mode of the display context.

```
typedef struct tagTEXTMETRIC {
LONG tmHeight;
LONG tmAscent;
LONG tmDescent;
LONG tmInternalLeading;
LONG tmExternalLeading;
LONG tmAveCharWidth;
LONG tmMaxCharWidth;
LONG tmWeight;
LONG tmOverhang;
LONG tmDigitizedAspectX;
LONG tmDigitizedAspectY;
TCHAR tmFirstChar;
TCHAR tmLastChar;
TCHAR tmDefaultChar;
TCHAR tmBreakChar;
BYTE tmItalic;
BYTE tmUnderlined;
BYTE tmStruckOut;
BYTE tmPitchAndFamily;
BYTE tmCharSet;
} TEXTMETRIC, *PTEXTMETRIC;
```

The Windows API **GetTextExtentExPoint** gets the number of characters in a string that will fit within a space.

```
BOOL GetTextExtentExPoint(
 HDC hdc,     // handle to DC
 LPCTSTR lpszStr, // character string
 int cchString,  // number of characters
 int nMaxExtent, // maximum width of formatted string
 LPINT lpnFit,  // maximum number of characters
 LPINT alpDx,   // array of partial string widths
 LPSIZE lpSize  // string dimensions
);
```

More Windows API Text Functions

The following Windows API functions are also useful for working with text:

- **CreateSolidBrush**
- **GetSysColor**
- **SetTextColor**
- **GrayString**

This section discusses these functions and shows examples of their use.

The **CreateSolidBrush** function creates a logical brush that has the specified solid color, as shown in the following example:

```
HBRUSH CreateSolidBrush(
COLORREF crColor // brush color value
);
```

The **GetSysColor** function retrieves the current color of the specified display element, as shown in the following example. Display elements are the parts of a window.

```
DWORD GetSysColor( int nIndex );
```

The **SetTextColor** function sets the text color for the specified device context to the specified color, as shown in the following example:

```
COLORREF SetTextColor(
HDC hdc, // handle to DC
COLORREF crColor // text color
);
```

The following example incorporates the use of **DrawText, CreateSolidBrush, GetSysColor,** and **SetTextColor**:

```
RECT myRectangle;
//+
// create a brush
//-
HBRUSH myBackgroundBrush =
CreateSolidBrush(
GetSysColor(COLOR_BACKGROUND) // color of system background
);
//+
// set the text color to the system's button text color
//-
SetTextColor(
hdc,
GetSysColor(COLOR_BTNTEXT) //color of text on buttons
);
// calculate myRectangle
//+
// fill in (erase) the area inside the rectangle with the
// system's background color
//-
FillRect( hdc, &myRectangle, myBackgroundBrush);
//+
// The DrawText function uses the device context's selected font, text
// color, and background color to draw the text. Unless the DT_NOCLIP
// format is used, DrawText clips the text so that it does not appear
// outside the specified rectangle.
//-
DrawText( hdc,
myString,
_tcsclen(myString), // use _tcsclen() vs. strlen()
&myRectangle,
(DT_CENTER | DT_SINGLELINE )
);
```

The **GrayString** function draws gray text at the specified location, as shown in the following example. The function draws the text by copying it into a memory bitmap, graying the bitmap, and then copying the bitmap to the screen. The function grays the text regardless of the selected brush and background. **GrayString** uses the font currently selected for the specified device context.

```
BOOL GrayString(
HDC hDC, // handle to DC
HBRUSH hBrush, // handle to the brush
GRAYSTRINGPROC lpOutputFunc, // callback function
LPARAM lpData, // application-defined data
int nCount, // number of characters
int X, // horizontal position
int Y, // vertical position
int nWidth, // width
int nHeight // height
);
```

Text Widgets and Controls

A text widget or control is used to display, enter, and edit text. The exact functionality of a text widget or control depends on how its resources are set.

X Windows example: Widget functionality

In X Windows, the widget functionality is set as shown in the following example:

```
text = XtVaCreateManagedWidget ( "myTextWidget",
asciiTextWidgetClass,
parentWidget,
XtNfromHoriz,
quit,
XtNresize,
XawtextResizeBoth,
XtNresizable,
True,
NULL);
```

Windows example: Widget functionality

In Windows API and GDI, the control functionality is set as shown in the following example:

```
//+
// Create an edit Control.
//-
HWND handleToThisEditControl;
handleToThisEditControl =
CreateWindow( TEXT("EDIT"), //ß the type of control
TEXT("Some Text"), //ß edit control text
(WS_CHILD |
WS_VISIBLE |
ES_READONLY |
ES_LEFT |
ES_UPPERCASE), //ß the control style
XpositionInParent,
yPositionInnParent,
CONTROL_WIDTH_IN_DEVICE_UNITS,
CONTROL_HEIGHT_IN_DEVICE_UNITS,
handleOfParentWindow, //ß parent window
(HMENU)NUMBER_USED_TO_ID_THIS_EDIT_CONTROL,
appContext,
NULL );
//+
// Turn off Read Only
//-
SendMessage( handleToThisEditControl ,
EM_SETREADOINLY,
(WPARAM)FALSE, //ß set read only false
(LPARAM)NULL );
//+
// set the edit control's text
//-
SetWindowText( handleToThisEditControl, TEXT("Some New Text") );
//+
// retrieve the edit control's text as text
//-
GetWindowText( handleToThisEditControl,
myStringBuffer,
myStringBufferMaxSize );
//+
// retrieve the edit control's text as an integer
//-
myIntegerValue =
GetDlgItemInt( handleOfParentWindow,
NUMBER_USED_TO_ID_THIS_EDIT_CONTROL,
&resultFlag, // did the translation succeed ?
FALSE ); // no this is an unsigned number
```

Property Sheets

A property sheet is a modal secondary window that allows the user to view and edit the properties of an item. You can also implement a property sheet as a modeless dialog. For example, property sheets can be used to display font and border properties for a worksheet, to set the properties of a device (such as a disk drive, printer, or mouse), or to display file system properties for a folder view.

A property sheet consists of a number of property pages. It is an instance of **CPropertySheet**.

After a property page is designed, while creating the class for this dialog box, ensure that the base class is **CPropertyPage**. It is attached to a property sheet using the **CPropertySheet::AddPage** function, as shown in the following code:

```
// CPage1, CPage2 and CPage3 are CPropertyPage derived classes
CPage1 p1;
CPage2 p2;
CPage3 p3(this);
CPropertySheet dlg;
dlg.SetTitle("Functions");
dlg.AddPage(&p1);
dlg.AddPage(&p2);
dlg.AddPage(&p3);

dlg.DoModal();
```

CPropertySheet::SetTitle can be used to set the title for the property sheet. And as seen in the previous code, a property sheet can be invoked using **DoModal.**

Toolbars

The toolbar is an object of the CToolBar class. Toolbars are control bars derived from the CControl Bar.

The toolbar is made up of a number of bitmaps, which can be painted using various tools. These bitmaps represent the toolbar buttons. Clicking them sends a command message. These buttons are associated with an ID, which can be viewed using the Properties window. The Properties window appears when you double-click the toolbar button. The prompt and ToolTip appear when the mouse passes over a button. The prompt is the message in the Status Bar.

The toolbar buttons and the menu items can be mapped to the same ID if their functionalities are the same, so that they can use the same handler.

The toolbar can be loaded using the **LoadToolBar** method of the CToolBar MFC class.

Update Command Handlers of Toolbar Buttons

Command handlers of toolbar buttons are similar to the Menu UpdateCommad handlers. The toolbars and status bars are displayed all the time and get updated during the idle time processing and during display of menu popups.

Update messages for toolbar buttons can be processed using the **CCmdUI::Enable member** function and the **CCmdUI::SetCheck** member function.

Status Bars

Status bars are control bars derived from the **CControlBar** class and can be created using the **CStatusBar** MFC class. Status bars are made up of a number of panes; the text for the panes is set using the **SetPaneText** function.

Following is the code to create a Status Bar, where *m_wndStatusBar* is an object of the MFC class **CStatusBar**.

```
static UINT indicators[] =
{
      ID_SEPARATOR,      // status line indicators
      ID_INDICATOR_CAPS,
      ID_INDICATOR_NUM,
      ID_INDICATOR_SCRL,
};
 if (!m_wndStatusBar.Create(this) ||
      !m_wndStatusBar.SetIndicators(indicators,
             sizeof(indicators)/sizeof(UINT)))  {
            TRACE0("Failed to create status bar\n");
            return -1;   // fail to create      }
```

Printing

This section discusses the APIs involved with printing in UNIX and Windows. You can use the information provided in this section to analyze various routines and APIs used in your UNIX applications and to identify the replacement implementation in the Windows environment using the Win32/Win64 API.

Printing Documents

The following topics discuss the various ways of printing documents in the UNIX and Windows environments.

Printing Using System Commands

In UNIX, system commands such as **lp** (in System V) and **lpr** (in BSD) are used to print documents. Windows provides the **print** system command for printing documents. Following is the syntax of **print** command:

```
PRINT [/D:device] [[drive:][path]filename[...]]
```

Where,

`/D:`*`device`*

Specifies a print device,

`drive`

Specifies the drive in which the file to be printed is located,

`path`

Specifies the path in which the file is located, and

`filename`

Specifies the exact name of the file to be printed with its extension.

The following command when entered at the MS-DOS® prompt prints the document called MyDocument.doc through the printer named MyPrinter.

```
PRINT /D:MyPrinter C:\MyDocument.doc
```

The **print** command can be executed by invoking the **system** command within application programs written in C/C++ or any other supporting language. Using the **system** command in an application for the earlier mentioned DOS command would be:

```
System("PRINT /D:MyPrinter C:\MyDocument.doc");
```

Printing Using APIs

X Windows libraries provide extended facilities for printing options in UNIX systems.

X Windows example: Printing a document

The following is sample code in X Windows that helps in printing a document.

```
#include <X11/Xlib.h>
#include <X11/extensions/Print.h>
main()
{
  Display *pdpy;
  Screen *pscreen;
  Window pwin;
  XPPrinterList plist;
  XPContext pcontext;
  int plistCnt;
  char *attrPool;
  #define NPOOLTYPES 5
  XPAttributes poolType[NPOOLTYPES] =
{XPJobAttr,XPDocAttr,XPPageAttr,XPPrinterAttr,XPServerAttr};
  int i;
  unsigned short width, height;
  XRectangle rect;
  char *printServerName = ":1";
  char *mylaser = "varos";

  /*
   * connect to the X print server
   */
  pdpy = XOpenDisplay( printServerName );

  /*
   * see if the printer "mylaser" is available
   */
  plist = XpGetPrinterList (pdpy, mylaser, &plistCnt );
  /*
   * Initialize a print context representing "mylaser"
   */
  pcontext = XpCreateContext( pdpy, plist[0].name );
  XpFreePrinterList( plist );
  /*
   * Possibly modify attributes in the print context
   */
  for(i=0;i < NPOOLTYPES;i++) {
  if(attrPool = XpGetAttributes( pdpy, pcontext, poolType[i] )) {
    /* twiddle attributes */
    /*
     XpSetAttributes( pdpy, pcontext, poolType[i],
              attrPool, XPAttrMerge );
    */
    XFree(attrPool);
  }
  }

  /*
   * Set a print server, then start a print job against it
   */
  XpSetContext( pdpy, pcontext );
  XpStartJob( pdpy, XPSpool );
  /*
   * Generate the first page
   */
  pscreen = XpGetScreenOfContext( pdpy, pcontext );
  XpGetPageDimensions( pdpy, pcontext, &width, &height,
       &rect);
  pwin = XCreateSimpleWindow( pdpy, RootWindowOfScreen( pscreen ),
        rect.x, rect.y, rect.width, rect.height, 2,
        BlackPixelOfScreen( pscreen),
        WhitePixelOfScreen( pscreen));
  XpStartPage( pdpy, pwin );
  /* usual rendering stuff..... */
  XpEndPage( pdpy );
  XpStartPage( pdpy, pwin );
  /* some more rendering.....  */
  XpEndPage( pdpy );
  /*
   * End the print job - the final results are sent by the X print
   * server to the spooler sub system.
   */
  XpEndJob( pdpy );
  XpDestroyContext( pdpy, pcontext );
  XCloseDisplay( pdpy );
}
```

Microsoft Windows implements device-independent display. In MFC, the **CView** class provides the basic functionality for user-defined view classes. The **OnDraw** member function of your view class can be used to perform screen display, printing, and print preview. For print preview, the target device is a simulated printer output to the display.

Following are the responsibilities of the view class:

- Inform the framework of the number of pages in the document.
- When asked to print a specified page, draw that portion of the document.
- Allocate and de-allocate any fonts or other GDI resources needed for printing.
- If necessary, send any escape codes needed to change the printer mode before printing a given page, for example, to change the printing orientation on a per-page basis.

Following are the responsibilities of the framework:

- Display the **Print** dialog box.
- Create a CDC object for the printer.
- Call the **StartDoc** and **EndDoc** member functions of the CDC object.
- Repeatedly call the **StartPage** member function of the CDC object, inform the **view** class which page should be printed, and call the **EndPage** member function of the CDC object.
- Call overridable functions in the view at the appropriate times.

In Windows programming, printing options are available with the following:

- Windows API
- MFC and ATL

Printing Using the Windows API

A Windows program can access a printer by calling the **PrintDlg** function. The **PrintDlg** function displays a **Print** dialog box. The **Print** dialog box enables the user to specify the properties of a particular print job. The general syntax of a **PrintDlg** function is as follows:

```
BOOL PrintDlg(PRINTDLG lppd);
```

The argument lppd is a pointer to a PRINTDLG structure that contains information used to initialize the dialog box. When **PrintDlg** returns, this structure contains information about the selections of the user.

The following sample code displays a **Print** dialog box so that the user can select options for printing a document. The sample code first initializes a PRINTDLG structure and then calls the **PrintDlg** function to display the dialog box. It sets the PD_RETURNDC flag in the Flags member of the PRINTDLG structure. This causes **PrintDlg** function to return a device context handle for the selected printer in the hDC member. You can use the handle to render output on the printer.

On input, the sample code sets the hDevMode and hDevNames members to NULL. If the function returns TRUE, these members return handles to DEVMODE and DEVNAMES structures containing the inputs from the user and information about the printer. You can use this information to prepare the output to be sent to the selected printer.

Windows example: Displaying a print dialog box

```
PRINTDLG pd;
HWND hwnd;

// Initialize PRINTDLG
ZeroMemory(&pd, sizeof(PRINTDLG));
pd.lStructSize = sizeof(PRINTDLG);
pd.hwndOwner  = hwnd;
pd.hDevMode  = NULL;   // Don't forget to free or store hDevMode
pd.hDevNames  = NULL;   // Don't forget to free or store hDevNames
pd.Flags    = PD_USEDEVMODECOPIESANDCOLLATE | PD_RETURNDC;
pd.nCopies   = 1;
pd.nFromPage  = 0xFFFF;
pd.nToPage   = 0xFFFF;
pd.nMinPage  = 1;
pd.nMaxPage  = 0xFFFF;

if (PrintDlg(&pd)==TRUE)
{
  // GDI calls to render output.

  // Delete DC when done.
  DeleteDC(pd.hDC);
}
```

Printing Using MFC and ATL

There are various options for invoking a **Print** dialog box with MFC programming. The most straightforward way is to create an object for the **CPrintDialog** class available in MFC and show a **Print** dialog box.

To use a CPrintDialog object

1. Create the object using the **CPrintDialog** constructor.
2. After the dialog box has been constructed, you can set or modify any value in the m_pd structure to initialize the values of the controls in the dialog box. The m_pd structure is of type PRINTDLG (which is same as that explained earlier).
3. After initializing the dialog box controls, call the DoModal member function to display the dialog box and allow the user to select print options. DoModal returns whether the user selected the OK (IDOK) or Cancel (IDCANCEL) button.

The following is the constructor of **CPrintDialog**:

```
CPrintDialog(
  BOOL bPrintSetupOnly,
  DWORD dwFlags = PD_ALLPAGES | PD_USEDEVMODECOPIES | PD_NOPAGENUMS |
PD_HIDEPRINTTOFILE | PD_NOSELECTION,
  CWnd* pParentWnd = NULL
);
```

Where,

`bPrintSetupOnly`

is set to TRUE to display the standard **Windows Print Setup** dialog box. And it is set to FALSE to display the **Windows Print** dialog box.

`dwFlags`

Provides one or more flags that can be used to customize the settings of the dialog box, combined using the bitwise OR operator.

`pParentWnd`

Provides a pointer to the parent or owner window of the dialog box.

The following sample displays a print dialog box and creates a printer device context from the DEVMODE and DEVNAMES structures:

```
// Display the Windows Print dialog box with "All" radio button
// initially selected. All other radio buttons are disabled.
CPrintDialog dlg(FALSE);
if (dlg.DoModal() == IDOK)
{
  // Create a printer device context (DC) based on the information
  // selected from the Print dialog.
  HDC hdc = dlg.CreatePrinterDC();
  ASSERT(hdc);
}
```

Table 6.4. Members of the CPrintDialog Class

Member Function	Description
HDC **CreatePrinterDC**()	Creates a printer device context without displaying the **Print** dialog box.
virtual INT_PTR**DoModal**()	Displays the dialog box and allows the user to make a selection.
int **GetCopies**() const	Retrieves the number of copies requested.
BOOL **GetDefaults**()	Retrieves device defaults without displaying a dialog box.
CString **GetDeviceName**() const	Retrieves the name of the currently selected printer device.
LPDEVMODE **GetDevMode**() const	Retrieves the DEVMODE structure.
CString **GetDriverName**() const	Retrieves the name of the currently selected printer driver.
int **GetFromPage**() const	Retrieves the starting page of the print range.
CString **GetPortName**() const	Retrieves the name of the currently selected printer port.
HDC **GetPrinterDC**() const	Retrieves a handle to the printer device context.
int **GetToPage**() const	Retrieves the ending page of the print range.
BOOL **PrintAll**() const	Determines whether to print all pages of the document.
BOOL **PrintCollate**() const	Determines whether collated copies are requested.
BOOL **PrintRange**() const	Determines whether to print only a specified range of pages.
BOOL **PrintSelection**() const	Determines whether to print only the currently selected items.

The header afxdlgs.h is to be included in case the user wants to use the **CPrintDialog** class in the program.

A simple mapping can be done between the methods available for printing in X Windows and those available in Windows, as listed in Table 6.5.

Table 6.5. Comparing X Windows and Windows Printing Options

X Windows	Windows	Description
XOpenDisplay	CString GetDeviceName() const	Retrieves the name of the currently selected printer device.
XPCreateContext	HDC CreatePrinterDC()	Creates a printer device context.
XPStartJob	int StartDoc LPDOCINFO lpDocInfo)	Informs the device driver that a new print job is starting.
	int StartDoc(LPCTSTR lpszDocName)	
XPEndJob	int EndDoc()	Ends a print job started by the previous member function.
XCreateSimpleWindow	virtual INT_PTR DoModal()	Displays the **Print** dialog box.
XPStartPage	int StartPage()	Informs the device driver that a new page is starting.

X Windows	Windows	Description
XPEndPage	int EndPage()	Informs the device driver that a page is ending.

Plotting Documents

The following topics discuss the various ways of plotting documents in UNIX and Windows environments.

Using the Plotters in UNIX

Using the UNIX command **lpr**, you can send a PostScript file to a plotter by using vcplt, vcpltg, or vcpltcf with the **-P** option.

For example, to send the file mygraphic.ps to vcplt, the command would be:

```
lpr -Pvcplt mygraphic.ps
```

By default, output is in color with normal or standard print quality. You can choose grayscale or specify print quality and paper size by using the **-X** option in the **lpr** command. The following are the available **-X** options:

- grayscale (or greyscale)
- pq=fast (or pq=draft)
- pq=normal
- pq=best
- paper=<width>x<height> (for example, paper=20x30 means 20 inches wide by 30 inches high)

For example, to choose grayscale and “best” print quality on vcplt, the UNIX command would be:

```
lpr -Pvcplt -Xgrayscale,pq=best filename.ps
```

Note that the paper option is rarely required because the application normally sets the paper size. If the application does not set the size, you can use the paper option to set it or accept the default paper size of 36" × 17."

Using the Plotters in Windows

The **PrinterSettings** class specifies information about how a document is printed, including the printer details that prints it. The following property of the **PrinterSettings** class returns a Boolean value, true or false, corresponding to whether the peripheral is a plotter or a raster printer:

```
public: __property bool get_IsPlotter();
```

This function can be used to identify a plotter.

Imaging

Imaging refers to how images are handled, loaded, saved, and stored in memory. This section discusses how imaging is done in UNIX and Windows.

Image Handling in UNIX

Xlib provides an image object that describes the data in memory and provides for basic operations on that data. You should reference the data through the image object instead of referencing the data directly. Supported operations include getting a pixel, storing a pixel, extracting a sub-image of an image, adding a constant to an image, and destroying the image.

```
typedef struct _XImage {
      int width, height;      /* size of image */
      int xoffset;      /* number of pixels offset in X direction */
      int format;       /* XYBitmap, XYPixmap, ZPixmap */
      char *data;       /* pointer to image data */
      int byte_order;         /* data byte order, LSBFirst, MSBFirst */
      int bitmap_unit;  /* quant. of scanline 8, 16, 32 */
      int bitmap_bit_order;   /* LSBFirst, MSBFirst */
      int bitmap_pad;         /* 8, 16, 32 either XY or ZPixmap */
      int depth;        /* depth of image */
      int bytes_per_line;     /* accelerator to next scanline */
      int bits_per_pixel;     /* bits per pixel (ZPixmap) */
      unsigned long red_mask; /* bits in z arrangement */
      unsigned long green_mask;
      unsigned long blue_mask;
      XPointer obdata;/* hook for the object routines to hang on */
      struct funcs {                /* image manipulation routines */
            struct _XImage *(*create_image)();
            int (*destroy_image)();
            unsigned long (*get_pixel)();
            int (*put_pixel)();
            struct _XImage *(*sub_image)();
            int (*add_pixel)();
      } f;
} XImage;
```

The functions **XInitImage**, **XPutImage**, **XGetImage**, and **XGetSubImage** are available in UNIX to initialize, put, get, or copy the image.

Image Handling in Windows

The **CImage** class, defined in the atlimage.h file that comes with MFC 7.0, provides enhanced bitmap support, including the capability to load and save images in Joint Photographic Experts Group (JPEG), Graphics Interchange Format (GIF), Bitmap (BMP), and Portable Network Graphics (PNG) formats.

After instantiating the object of the **CImage** class, call **Create**, **Load**, **LoadFromResource**, or **Attach** to attach a bitmap to the object. Operator HBITMAP of the **CImage** class returns the Windows handle attached to the **CImage** object.

In addition, the operations listed in Table 6.6 are also possible with the **CImage** class.

Table 6.6. Member Functions of CImage Class

Member Function	Description
AlphaBlend	Displays bitmaps that have transparent or semitransparent pixels.
Attach	Attaches an **HBITMAP** to a **CImage** object. Can be used with either non-DIB section bitmaps or DIB section bitmaps.
BitBlt	Copies a bitmap from the source device context to this current device context.
Create	Creates a DIB section bitmap and attaches it to the previously constructed **CImage** object.
CreateEx	Creates a DIB section bitmap (with additional parameters) and attaches it to the previously constructed **CImage** object.
Destroy	Detaches the bitmap from the **CImage** object and destroys the bitmap.
Detach	Detaches the bitmap from a **CImage** object.
Draw	Copies a bitmap from a source rectangle into a destination rectangle. **Draw** stretches or compresses the bitmap to fit the dimensions of the destination rectangle, if necessary, and handles alpha blending and transparent colors.
GetBits	Retrieves a pointer to the actual pixel values of the bitmap.
GetBPP	Retrieves the bits per pixel.
GetColorTable	Retrieves red, green, blue (RGB) color values from a range of entries in the color table.
GetDC	Retrieves the device context into which the current bitmap is selected.
GetExporterFilterString	Finds the available image formats and their descriptions.
GetHeight	Retrieves the height of the current image in pixels.
GetMaxColorTableEntries	Retrieves the maximum number of entries in the color table.
GetPitch	Retrieves the pitch of the current image in bytes.
GetPixelAddress	Retrieves the address of a given pixel.
GetPixel	Retrieves the color of the pixel specified by the x and y axes.
GetTransparentColor	Retrieves the position of the transparent color in the color table.
GetWidth	Retrieves the width of the current image in pixels.

Member Function	Description
IsDibSection	Determines if the attached bitmap is a DIB section.
IsIndexed	Indicates that the colors of a bitmap are mapped to an indexed palette.
IsNull	Indicates if a source bitmap is currently loaded.
IsTransparencySupported	Indicates whether the application supports transparent bitmaps and was compiled for Windows 2000 or later.
LoadFromResource	Loads an image from the specified resource.
Load	Loads an image from the specified file.
MaskBlt	Combines the color data for the source and destination bitmaps using the specified mask and raster operation.
PlgBlt	Performs a bit-block transfer from a rectangle in a source device context into a parallelogram in a destination device context.
ReleaseDC	Releases the device context that was retrieved with CImage::GetDC.
ReleaseGDIPlus	Releases resources used by GDI+. Must be called to free resources created by a global **CImage** object.
Save	Saves an image as the specified type. **Save** cannot specify image options.
SetColorTable	Sets red, green, blue (RGB) color values in a range of entries in the color table of the DIB section.
SetPixelIndexed	Sets the pixel at the specified coordinates to the color at the specified index of the palette.
SetPixelRGB	Sets the pixel at the specified coordinates to the specified red, green, blue (RGB) value.
SetPixel	Sets the pixel at the specified coordinates to the specified color.
SetTransparentColor	Sets the index of the color to be treated as transparent. Only one color in a palette can be transparent.
StretchBlt	Copies a bitmap from a source rectangle into a destination rectangle, stretching or compressing the bitmap to fit the dimensions of the destination rectangle, if necessary.
TransparentBlt	Copies a bitmap with transparent color from the source device context to this current device context.

Additional information on using the **CImage** class is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcsample/html/vcsamSimpleImageSample.asp.

Table 6.7 lists the mapping that helps in the analogous study of the image-processing concepts in X Windows and Windows programs.

Table 6.7. Mapping Between X Windows and Windows Imaging Options

X Windows	Windows	Description
XReadBitmapFile	Attach	Reads in a file containing a bitmap into the object.
XGetImage	BitBlt	Copies a bitmap from the source device context to this current device context.
XCreateImage	Create	Creates a bitmap and attaches it to the previously constructed image object.
XDestroyImage	Destroy	Detaches the bitmap from the image object and destroys the bitmap.
XPutImage	Draw	Copies a bitmap from a source rectangle into a destination rectangle.
XFindContext	GetDC	Retrieves the device context into which the current bitmap is selected.
XGetPixel	GetPixel	Retrieves the color of the pixel specified by the x and y axes.
XDeleteContext	ReleaseDC	Releases the device context for the given resource ID.
XPutPixel	SetPixel	Sets the pixel at the specified coordinates to the specified color.

Mapping X Windows Terminology to Microsoft Windows

The graphical models of UNIX and Microsoft Windows are very different. There are conceptual similarities but little side-by-side mapping is possible. This section describes as many connections as possible. In the headings of this section, the X Windows term is followed by the corresponding Windows GDI term in the following format:

X Windows term vs. *Windows term*

This section enables you to map various X Windows terminologies used in your application to the corresponding Windows terminologies.

Callback vs. WindowProc

Windows uses the **WindowProc** function in the same capacity as **Callback** in X Windows. An X widget can have a list of callbacks associated with it, but in Windows, a window has a single entry point for handling messages sent to it.

Client vs. Client Window

X Windows comprises a protocol that describes how a client interacts with a server that could be running on a remote computer. How objects are drawn is the responsibility of the server. This provides device-independence for the client application because it is not responsible for knowing anything about the physical hardware. In the Microsoft Windows environment, the graphics device interface (GDI) API provides this layer of device-independence. Windows-based applications, such as X clients, are not required to access graphics hardware directly. GDI interacts with the hardware by using device drivers on behalf of the application.

A single Windows-based application can contain any number of separate windows. Each of these can have a window frame, caption bar, system menu, minimize and maximize buttons, and its own main display area, which is referred to as the client window.

In Windows, multiple document interface (MDI) applications have three kinds of windows. These are a frame window, an MDI client window, and a number of child windows. The term client window takes on a special meaning in this case. For more information about MDI, see the MDI documentation on the MSDN Web site or the Platform SDK.

Console Mode vs. Command Window

If X Windows or some other graphical user interface is not running on a UNIX system, a user must work in text only or in console mode. Microsoft Windows is exactly the opposite. If a console is not running, the user must work in GUI mode. Windows text-based mode is provided by running the **Cmd.exe** tool. This environment is also referred to as a command window or the MS-DOS prompt. To run the **Cmd.exe** tool, click **Start**, click **Run**, in the **Run** dialog box type **cmd**, and then click **OK**. Developers can also use the Console API to build native Windows API-based console applications. For more information, search for “Console Functions” on the MSDN Web site.

DPI vs. Screen Resolution

When starting an X Windows session, using the -dpi (dots per inch) option can improve appearance on displays with larger resolutions, such as 1600 × 1200. The -dpi option also helps to work around possible font issues. A Windows-based application is usually built with no assumptions about the capabilities of the system it will be running on. System APIs are used to calculate proper scaling and other characteristics. **GetDeviceCaps** is used to obtain the DPI of the system. **GetSystemMetrics** and **SystemParametersInfo** provide information about practically every graphical element needed to calculate sizes for fonts and other graphical elements. For more information, search for “dots per inch” on the MSDN Web site.

Graphics Context vs. Device Context

The X Windows graphics context contains required information about how drawing functions are to be executed. The Windows API device context provides similar information. The functions used in each are listed in Table 6.8.

Table 6.8. X Windows Graphics Context and Windows API Device Context Comparable Functions

Xlib	Windows
XtGetGC	GetDC
XtReleaseGC	ReleaseDC
XCreateGC	CreateDC
XFreeGC	DeleteDC

For more information about using graphics context and device context, see “Graphics Device Interface” earlier in this chapter.

Resources vs. Properties

In X Windows terminology, a widget is defined by its resources. Width, height, color, and font are examples of resources. Resources can be managed by using the **XtVaCreateManagedWidget** method or by using resource files or **XtVaGetValues** and **XtVaSetValues** functions.

In Windows terminology, a control is defined by its properties. For example, a text control has the Center Vertically, No Wrap, Transparent, Right Aligned Text, and Visible properties.

Resource Files vs. Registry

X Windows systems use configuration files referred to as resource files to store information about system settings or preferences for a particular X Windows client. In a Windows-based system, this type of information is stored in the registry. The registry stores data in a hierarchically structured tree. The Windows API has more than 40 functions to help access the registry. For more information, search for "registry" or "registry functions" on the MSDN Web site.

Resource file can take on another meaning in Windows-based application development.

Root Window vs. Desktop Window

All X Windows windows are descendents of the root window. In the Windows environment, the desktop window is a system-defined window that is the basis for all windows displayed by all applications.

/bin vs. /System32

In Windows, the /System32 directory is roughly equivalent to the /bin directory on a UNIX system. This is where the system executable files are located. The /System32 directory is located in the system root directory. To find system root, at the command prompt type **set** and press ENTER. This displays a listing of the current environment. In the list, locate SYSTEMROOT. Under SYSTEMROOT, there is an entry similar to SYSTEMROOT=C:\WINNT. This is the system directory, and under this directory is the /System32 directory.

/usr/bin vs. Program Files

The Program Files directory on a Windows-based system is similar to the /usr/bin directory on a UNIX system. This is a default location for user applications. In Windows, a user can create more than one such directory. Each drive, for example, has a Program Files directory. The system environment variable ProgramFiles contains the path of one default location, for example,

ProgramFiles=C:\ProgramFiles.

/usr/lib vs. LIB Environment Variable

In Windows, the path to user libraries can be to anywhere. To manage this relationship, retrieve or set the system environment variable LIB.

/usr/include vs. INCLUDE Environment Variable

In Windows, the path to user include files may be to anywhere. To manage this relationship, retrieve or set the system environment variable INCLUDE.

Pixmap (or Bitmap) vs. Bitmap

In X Windows, bitmap and pixmap have the same usage as Windows bitmaps. For example, they can be used as pictures, fill patterns, icons, and cursors. They are, however, very different in form.

X Windows example: A 16 × 16 "X" figure

The following X Windows example represents a simple 16 × 16 "X" figure.

```
#define simple_width 16
#define simple_height 16
static unsigned char simple_bits[] = {
0x01, 0x80, 0x02, 0x04, 0x20, 0x08, 0x10, 0x10, 0x08, 0x20, 0x04,
0x40, 0x02, 0x80, 0x01, 0x80, 0x01, 0x02, 0x20, 0x04, 0x10, 0x08,
0x08, 0x10, 0x04, 0x20, 0x02, 0x40, 0x01, 0x80
};
```

Windows example: A 16 × 16 "X" figure

The following Windows example also represents a simple 16 × 16 "X" figure.

```
000000 42 4D 7E 00 00 00 00 00 00 00 3E 00 00 00 28 00
000010 00 00 10 00 00 00 10 00 00 00 01 00 01 00 00 00
000020 00 00 40 00 00 00 CA 0E 00 00 C4 0E 00 00 00 00
000030 00 00 00 00 00 00 00 00 00 00 FF FF FF 00 7F FE
000040 00 00 BF FD 00 00 DF FB 00 00 EF F7 00 00 F7 EF
000050 00 00 FB DF 00 00 FD BF 00 00 FE 7F 00 00 FE 7F
000060 00 00 FD BF 00 00 FB DF 00 00 F7 EF 00 00 EF F7
000070 00 00 DF FB 00 00 BF FD 00 00 7F FE 00 00
```

Window Manager vs. Windows Server 2003 and Windows XP

A special kind of X Windows client, called the Window Manager, provides a consistent working environment in the root window.

In a Microsoft Windows environment, the operating system itself is the window manager and it provides the desktop window. When a user logs on, the system creates three desktops within the WinSta0 windows station. For more information, search for "WinSta0" on the MSDN Web site.

Widgets are usually represented as controls in Windows API-based applications. Like the X Windows environment, the Windows API offers many widgets to choose from, and a great number of third-party versions are also available. Sometimes deciding exactly what to call which is difficult. For example, X Windows dialog boxes are widgets. In the Windows API, however, dialog boxes are not considered to be controls, although objects such as dialog boxes, buttons, and scroll bars are all windows.

X Library [Xlib] [X11] vs. Gdi32.lib

The X Windows library [Xlib][X11] is the lowest level library. Like Gdi32.lib, it provides all the basic drawing functions.

X Toolkit [Intrinsics] [Xt] vs. User32.lib

The X Toolkit (Xt) is a library that accesses the lower-level graphics functionality of Xlib (X Windows) and provides such user interface elements as menus, buttons, and scroll bars. It is similar to User32.lib except that in the Windows environment, the look and feel of widgets or controls is provided in User32.lib instead of by higher-level libraries.

Porting OpenGL Applications

OpenGL was originally developed by Silicon Graphics as a platform-independent set of graphics APIs. This has made OpenGL an attractive option for developers who want to target multiple platforms. Very little, if any, platform-specific code is required to move a graphics application from one platform to another. OpenGL extensions enable the segregation and handling of platform-specific code.

OpenGL is not, however, a set of windowing libraries. An OpenGL application with Windows uses either the windowing system of the target platform (X Windows or Windows), or a cross-platform library such as the OpenGL Graphics Library Utility Kit (GLUT). Because of licensing concerns, however, most commercial applications incorporate the target platform windowing system. Therefore, when moving a UNIX application that uses OpenGL to Windows, migration considerations similar to those for a non-OpenGL application are likely to apply. This section covers additional GUI considerations for the migration of OpenGL applications.

In addition to the windowing system itself, OpenGL applications require a context to the host windowing system. A special set of OpenGL extensions for window context have been developed. UNIX applications typically use the GLX OpenGL extensions for X Windows. Microsoft Windows-based applications typically use the WGL (wiggle) OpenGL extensions. In either case, three main functions are required:

- **Create context**
    - X Windows: glXCreateContext
    - Windows: **wglCreateContext**
- **Make context current**
    - X Windows: **glXMakeCurrent**
    - Windows: **wglMakeCurrent**
- **Delete context**
    - X Windows: **glXDeleteContext**
    - Windows: **wglDeleteContext**

OpenGL contains no equivalents for the IRIS GL text-handling calls and Font Manager calls. To obtain facilities for handling full text and font, GLX OpenGL extensions for X Windows API **glXUseXFont** is used. This API generates a series of display lists, one for each character in the font.

The equivalent API in WGL OpenGL extension are **wglUseFontBitmaps**/¶**wglUseFontOutlines**.

The **wglUseFontBitmaps** API creates a set of bitmap display lists for use in the current OpenGL rendering context. The set of bitmap display lists is based on the glyphs in the currently selected font in the device context. Use the bitmaps to draw characters in an OpenGL image.

The **wglUseFontOutlines** API creates a set of display lists, one for each glyph of the currently selected outline font of a device context, for use with the current rendering context. The display lists are used to draw 3-D characters of TrueType fonts. Each display list describes a glyph outline in floating-point coordinates.

The standard GDI font and text drawing functions draw text in a single-buffered OpenGL window. These functions cannot be used to draw text in a double-buffered OpenGL window. To draw text in a double-buffered OpenGL window, an OpenGL display list for bitmap images of characters must be built and then be executed.

To draw text in a double-buffered OpenGl window

1. Select a font for a device context, setting the properties of the font as required.
2. Create a set of bitmap display lists based on the glyphs in the font used by the device content, one display list for each glyph that the application will draw.
3. Draw each glyph in a string, using those bitmap display lists.

The **wglUseFontBitmaps** and the **wglUseFontOutlines** API is used for creating the display lists.

For more information about fonts in OpenGL, refer to the operating system Help documentation or search the MSDN Web site.

After the migration, if the application still needs support for UNIX, use the C/C++ precompiler directives (#ifdef) to target the appropriate platform.

The OpenGL API is a C library on both UNIX and Windows. Fortran applications can also use OpenGL. To make it easier for Fortran applications to use OpenGL, a Fortran 90 Module often exists to handle the translation between Fortran and C-calling conventions. Most Fortran compilers on Windows provide an optional Fortran module for OpenGL.

For more information about OpenGL and platform-specific examples, refer to the following Web sites:

- The SGI OpenGL Web site, available at http://www.sgi.com/software/opengl/.
- The OpenGL Web site, available at http://www.opengl.org/.

Chapter 7: Developing Phase: Migrating Fortran Code

This chapter examines the process of migrating a Fortran code to the Microsoft® Windows® operating system from the UNIX environment. This migration may be to either the Windows subsystem or the Interix subsystem.

Fortran is relatively easy to port between platforms, largely because of the high degree of standardization between Fortran compilers and the types of applications for which Fortran is typically used.

The main difficulties with migrating Fortran code are often associated with either the integration of an application with other languages or its use of third-party libraries.

This chapter covers how you should:

- Gather and analyze data regarding your Fortran application.
- Select development tools and other resources for developing the Fortran code in the Windows environment.
- Design and validate your Fortran migration.
- Plan the migration of your Fortran application—specifically how to port the UNIX Fortran application to Windows using the Windows API.
- Use Microsoft Visual Studio® .NET to debug the Fortran application.

Data Gathering and Analysis

This section discusses the range of data you need to gather when you migrate Fortran code so that you can determine the best migration approach. The data that you must collect includes the following:

- The Fortran level used (77, 90, or 95).
- The graphical user interface (GUI) requirements.
- Any required third-party libraries.
- Whether the application provides any cross-language support.
- The best development and build environment to use for the migration.

In addition, you must know how your Fortran code is being used, for example:

- Is it a stand-alone application?
- Does it expose interfaces for other languages?
- Does it call interfaces in other languages?

Fortran is commonly used in computationally intensive applications. Often, this means that Fortran is used in loosely coupled, high-performance grid computing environments. Typically, distributed grid computing requires integration with dynamic scheduling and high-performance message passing. In most cases, Fortran modules are not a part of the core infrastructure services; they must therefore integrate with libraries that perform these functions.

It is also important to understand the target development environment for the Fortran migration. Possible target environments include:

- Migration of the development environment from UNIX to Windows and the Windows application programming interface (API).
- Continued development on UNIX with Windows as a cross-platform port.
- Migration of the development environment from UNIX to a UNIX style development environment on Windows, such as Microsoft Interix.

These considerations must be addressed along with the actual source code migration.

Using Third-Party Libraries (Mixed Languages)

In addition to running as stand-alone modules, Fortran modules can call subprograms or be called by programs from other languages. This is true even for those Fortran modules that are contained within a dynamic-link library (DLL) or a static library.

This section describes mixed language compatibility with Fortran for both managed code and unmanaged code. Managed code is architecture-independent code that runs under the control of the Microsoft .NET common language runtime (CLR) environment. Unmanaged code is native, architecture-specific code.

Mixed-language applications can supply main programs and subprograms in the following formats:

- Compiled objects (.obj) and static libraries (.lib)
- .dll
- .NET managed code assembly

When using mixed languages with Fortran, you must resolve the issues that are caused by differences in the following:

- Calling conventions, which specify how arguments are moved and stored.
- Naming conventions, which specify how symbol names are altered when placed in an .obj file.

The need to address differences in calling and naming conventions between Fortran and C or C++ is not unique to a migration from UNIX to Windows. The Windows platform offers a wide range of third-party libraries to perform functions that can eliminate the need of some of the UNIX-based custom code. The use of these types of third-party C or C++ libraries in Windows eases the overall migration task and is an advantage in migrating to Windows.

This section focuses on integrating Fortran with C and C++ libraries in the Windows subsystem. It begins by examining the default calling and naming conventions for Fortran and how these conventions work with typical Windows C and C++ libraries and with Windows APIs.

Table 7.1 lists the default calling and naming conventions for Fortran, C and C++, and Windows APIs.

Table 7.1. Fortran, C or C++, and Windows API Calling and Naming Conventions

Source Type	Arguments	Procedure Case	Stack Cleanup	Argument Suffix
Fortran	By reference	Procedure name in all uppercase	The procedure being called is responsible for removing arguments from the stack before returning to the caller.	Yes
C/C++	By value	Procedure name in all lowercase	The procedure doing the call is responsible for removing arguments from the stack after the call is finished.	No
Windows API (**STDCALL**)	By value	Procedure name in all lowercase	The procedure being called is responsible for removing arguments from the stack before returning to the caller.	Yes

The next two sections discuss in detail the issues related to calling and naming conventions.

Calling Conventions

During a migration, a developer usually encounters both the C or C++ calling convention and the **STDCALL** convention. Some of the common development libraries can take care of some of the differences in calling conventions. Other libraries require explicit declarations.

Differences in calling conventions can be handled in a number of ways, including the use of:

- The Fortran **Interface** statement.
- C function declarations.
- Compatibility layers.
- Modular code that uses Fortran libraries.

These are explained in the next sections.

The Fortran Interface Statement

You can use the Fortran **Interface** statement to transform Fortran calling conventions into C style conventions. The following example takes a call to the C function, **CLibFunction**, which calls a C function that passes an integer value. An alias attribute is used to take care of the difference in case and the extra underscore.

```
INTERFACE
 SUBROUTINE CLIBFUNCTION(I)
   !MS$ATTRIBUTES C, ALIAS:'_CLibFunction' :: CLIBFUNCTION
   INTEGER I
 END SUBROUTINE CLIBFUNCTION
END INTERFACE
```

Attributes can be defined with the **Interface** statement to adjust the Fortran calling conventions so that they match the existing C libraries.

C Function Declarations

You can also use a combination of the C function declarations, function naming, and argument definitions to resolve calling convention differences. You can do this using the **STDCALL** or C options.

In C and C++ modules, you can specify the **STDCALL** calling convention by using the __stdcall keyword in a function prototype or definition. Windows procedures and API functions also use the __stdcall convention. The following example shows how a **STDCALL** function declaration handles stack calling conventions and suffixes in a manner similar to default Fortran conventions:

```
extern “C” void __stdcall FLIBUNCTION (int n);
```

This type of translation is especially useful when dealing with C++ name mangling (where the compiler adds characters to function names). Implementing the **STDCALL** convention tells the compiler that this function is not subject to C++ name mangling.

Alternatively, instead of changing the calling convention of the C code, using the Intel Fortran compiler, you can adjust the Fortran source code by using the C option. This is set with the ATTRIBUTES directive. For example, the following declaration assumes the subroutine is called with the C calling convention:

```
SUBROUTINE CALLED_FROM_C (A)
  !DEC$ ATTRIBUTES C :: CALLED_FROM_C
  INTEGER A
```

Compatibility Layers

Another means of resolving calling convention differences is to use a compatibility layer to translate between Fortran and C or C++. Using this approach, C and C++ libraries can expose **STDCALL** type interfaces while actually calling the C or C++ routines using C calling conventions.

The strategy you use for third-party library integration also depends on whether the Fortran or C or C++ code can be modified. For example, introducing Fortran **Interface** statements is only a viable option if the developer can modify the Fortran source code. The same is true regarding changing of function declarations in C or C++ source code.

This makes the concept of a compatibility layer the most flexible solution. However, this solution requires you to develop and maintain additional source code.

Fortran Modules

As with most languages, modularity allows for easier cross-platform development. Because Fortran is most often used for high-performance computations, platform-specific routines may already exist in external, non-Fortran routines.

Even though Fortran is often isolated to computationally specific routines, platform-specific APIs, such as threading, synchronization, and GUI functions, can also be used in Fortran. The Windows APIs for threading, synchronization, and GUI functions are typically made available using Fortran modules.

For example, Fortran modules can exist for Windows GUI functions, threading, and OpenGL graphics. The Fortran modules encapsulate the C and **STDCALL** style functions to the Windows kernel and Windows API libraries. To enhance portability, platform-specific code should either be encapsulated in Fortran modules or through a call layer to C and C++ functions.

The Fortran module feature requires a Fortran 90 or later compiler.

Fortran GUI applications often use OpenGL for high-performance graphics. OpenGL provides a cross-platform API for GUI development with minimal platform-specific code requirements. In Windows, OpenGL is a C library and can be used either from a Fortran module or through a custom OpenGL GUI extraction layer.

Your migration choice will largely depend on whether the existing UNIX Fortran application extracts the GUI calls or uses a Fortran module. Either strategy can be migrated to Windows.

Naming Conventions

Names are an issue for external data symbols shared among parts of the same program as well as among external routines. Symbol names, such as the name of a subroutine, identify a memory location that must be consistent among all calling routines.

Parameter names (names given in a procedure definition to variables that are passed to it) are never affected.

Reasons for altering names include:

- Case sensitivity (for example, in C and Macro Assembler (MASM)) or lack of case sensitivity (for example, in Fortran).
- Name decoration (for example, in C++).

If naming conventions are not reconciled, the program cannot successfully link and you will receive an "unresolved external" error.

The following list summarizes how to reconcile names between languages:

- **All-uppercase names**. If you call a Fortran routine that uses Fortran defaults and cannot recompile the Fortran code, then in C, you must use an all-uppercase name to make the call. Use of just the __stdcall convention in C code is not enough because __stdcall and **STDCALL** always preserve case in these languages. Fortran generates all-uppercase names by default and the C code must match it.

  For example, these prototypes establish the Fortran function **FFARCTAN**(angle), where the argument angle has the ATTRIBUTES VALUE property:

  ```
  In C:
  ```

  ```
  extern float __stdcall FFARCTAN( float angle );
  ```
- **All-lowercase names**. If the name of the routine appears as all lowercase in C, naming conventions are automatically correct when the C or STDCALL option is used in the Fortran declaration. This is because any case, including mixed case, may be used in the Fortran source code, and the C and STDCALL options change the name to all lowercase.
- **Mixed-case names**. If the name of a routine appears as mixed-case in C or MASM and if you cannot change the name, then you can resolve this naming conflict by using the Fortran ATTRIBUTES ALIAS option. ALIAS is required in this situation because Fortran will otherwise not preserve the mixed-case name.

To use the ALIAS option, place the name in single quotation marks exactly as it is to appear in the .obj file.

The following is an example for referring to the C function **My_Proc** on IA-32 systems:

```
!DEC$ ATTRIBUTES ALIAS:'_My_Proc' :: My_Proc
```

On Itanium-based systems, the same example would be coded without the leading underscore as:

```
!DEC$ ATTRIBUTES ALIAS:'My_Proc' :: My_Proc
```

To make this example work on both IA-32–based and Itanium-based systems, use the following:

```
!DEC$ ATTRIBUTES DECORATE,ALIAS:'My_Proc' :: My_Proc
```

Using Intel Fortran for Calling Non-Fortran Subprograms

A Fortran main program or subprogram can call a non-Fortran subprogram by using the Intel Fortran Compiler, supplied as one of the following:

- **Compiled object or static library**. Name .obj or .lib in Project->Properties->Linker->Input->Additional Dependencies or on command line that links the application.

  If creating a mixed-language solution in the Microsoft Visual C++® IDE, make the other project a Win32® static library and make it a dependent project of the Intel Fortran project. The library will get linked in automatically.
- **DLL**. Supply import library, if available, as done in compiled object for static library.

  If there is no import library, use Windows API routines **LoadLibrary** and **GetProcAddress** and call the procedure through an integer pointer.
- **.NET managed code assembly**. Use the Intel Fortran Module Wizard to generate the interfaces for routines in the assembly.

A Fortran subprogram can be called by a non-Fortran main program or subprogram as a compiled object, static library, or dynamic-link library, depending on the capability of the calling language. When calling Fortran code from a .NET managed code assembly, the Fortran code must be in a DLL.

Note Additional information on the Intel Fortran compiler is available at http://www.intel.com/software/products/compilers/fwin/whatsnew.htm.

Integrating Fortran with POSIX Applications

The Fortran application that you are migrating may be required to integrate with other POSIX-style applications. In this situation, the target Windows environment can be either the Windows POSIX subsystem (Interix), or a UNIX emulator running on the Windows subsystem. Microsoft Interix is the full-featured POSIX subsystem on Windows. MKS NuTCRACKER and Cygwin are examples of UNIX emulators.

The Fortran considerations for using either the Interix POSIX subsystem or a UNIX emulator are the same as for C or C++ migrations.

Note As of the time of publication, the GNU Fortran 77 compiler, f77, is the only Fortran compiler available for Interix.

Development Tools and Resources

This section provides you with information on the various development tools and resources such as compilers and IDEs for developing the Fortran code in the Windows environment. Microsoft supplies the GNU Fortran 77 compiler with the Interix subsystem. Microsoft does not supply or sell a Fortran compiler for Windows. For migrations that require Fortran 90 or Fortran 95 features, a third-party Fortran compiler, such as the Intel Visual Fortran compiler or the Lahey/Fujitsu Fortran compiler, is required to target Windows.

When you migrate Fortran applications, you must ensure that you implement the necessary development tools, including a source-code control system and build analysis and management tools. You should also consider your cross-platform build and debug environments. The Fortran version that you use for development must be compatible with the other development tools that you use during the migration.

For example, if your migration targets the Microsoft Visual Studio® .NET 2003 development system, your Fortran compiler should integrate with Visual Studio .NET 2003. The Intel Fortran Compiler 8.1 can plug into the Visual Studio .NET 2003 development environment.

Design and Validation

When you migrate a Fortran application from UNIX to Windows, the design and capabilities of the Fortran code must be an integral part of the design of your migrated application. This section will help you to estimate the effort required for migration and to identify potential risks in the Fortran migration. Considerations such as performance, library interoperability, and feature set can determine the overall success of the project.

Sizing the Fortran Migration

The effort required to conduct a Fortran migration depends largely on the answers to the following questions:

- Is the code modular?
- Will platform-specific code need migration within Fortran?
- What third-party libraries will Fortran code need and are these compatible with Windows?
- Is the Fortran module calling any non-Fortran language function/module?
- What is the version of the Fortran compiler?
- Is GUI or graphics support required?
- Do feature or function abstraction layers already exist in UNIX?

Because it is likely that the features and functions needed for a Windows migration already exist on UNIX, answers to these questions might already be available. If the code is already modular with feature or function abstraction layers, the code itself will move easily across as a port to Windows. In this case, the bulk of your effort will be spent in choosing any required third-party libraries, the cross-language calling conventions, and the integrated development environment (IDE) tools.

Assessing and Mitigating Risk

Fortran adds complexity and, consequently, risk to a migration because of the following reasons:

- You might require a third-party Fortran compiler.
- You might need call-level integration between the Fortran and C or C++ code.
- You will need a cross-language build and debug strategy for Windows.

You can mitigate Fortran migration risks by:

- Defining the Windows development environment, including the Fortran compiler and your integration strategy for C or C++ code and third-party libraries.
- Implementing modularity of the Fortran code and putting platform-specific features into a C or C++ compatibility layer. This will enhance the capability of the code to migrate from UNIX to Windows and is essential if the application needs to target both the UNIX and Windows platforms.
- Identifying the tools for building and debugging Fortran code and other interfacing languages like C or C++ in the Windows environment.

Migration Planning

This section describes how you should plan a Fortran migration and discusses several migration strategies. In particular, this section provides information about how to scope a Fortran migration to Windows using the Windows API.

Scoping the Fortran Migration

At first glance, an ANSI Fortran application migration can have most (if not all) of the same migration combinations as a C or C++ migration. A Fortran application can:

- Be multiuser.
- Have a GUI for user interaction.
- Use platform-specific features.

However, most Fortran applications perform specialized functions where the Fortran language is particularly suitable. For example, Fortran is particularly suited as a language for computationally intensive mathematical operations. Other languages, such as C and C++, provide more widely used features and libraries for such things as process and thread support and GUI features. For this reason, Fortran source code often performs only the computationally intensive functions in an application, leaving the process management and user interaction to C and C++.

Using Fortran in this manner removes most of the platform-specific issues from an ANSI Fortran migration. This means that in most cases, you can port the Fortran code of an application from UNIX to the Windows API with minimal changes. The C or C++ code usually requires the majority of the migration effort.

Porting Fortran to Interix

Before rewriting a Fortran application for Windows, you should consider other migration strategies. Porting to Interix represents just one possible strategy.

For porting UNIX style source to Interix, the GNU Fortran 77 compiler is provided. As the level is Fortran 77, applications that require Fortran 90 support, such as module support, cannot be ported to Interix. In addition, because POSIX subsystem libraries cannot be mixed with Windows subsystem libraries, Windows API versions of C and C++ that require Fortran libraries cannot use them from Interix. This leaves stand-alone Fortran 77 applications that interoperate with **stdin** and **stdout** as the best candidates for a UNIX style port to Interix.

Porting UNIX Fortran Source to Windows Using the Windows API

This section describes how to port Fortran code to Windows. Most Fortran migrations are a port, not a rewrite, to Windows. However, Fortran migrations involve migrating and integrating other language modules in the application. Techniques and strategies for using C and C++ tools and source from Fortran are required to complete the application migration.

For this purpose, most of the discussion in this section focuses on how to integrate Fortran code with C and C++ modules or libraries on the Windows platform.

Using C or C++ Libraries or Fortran Modules

Fortran applications can access cross-platform libraries either by using C and C++ libraries or through Fortran modules (available with Fortran 90 and later versions). If the cross-platform libraries are written in C or C++, there is little (if any) difference between this type of strategy and a port-to-Windows strategy. Currently, there are few third-party Fortran module suppliers. This is because of the limited market for Fortran and the fact that Fortran compilers are provided by third parties. Microsoft does not provide a Fortran compiler. Fortran modules are typically supplied by the compiler vendor or are created in-house.

Porting Fortran to Windows

It is possible to rewrite an application written entirely in Fortran to target Windows. The key task here is to identify how platform-specific features are implemented in Fortran. This is typically done with Fortran modules. If this is the case, you must identify or develop a corresponding Fortran module for each feature on Windows that exists on the source platform. For example, if an OpenGL module and threading module are used on UNIX, you must identify or develop a corresponding OpenGL and threading module to use on Windows.

Debugging Fortran Using Visual Studio .NET 2003

Although your Fortran application must be developed in an environment outside Visual Studio .NET 2003, you may need to include your Fortran code as a library or object module in a C or C++ project in Visual Studio .NET 2003. In most cases, this requires the Visual Studio .NET 2003 debugger to step through the Fortran code, as well as the C and C++ codes. This section explains how you can integrate Fortran libraries and object modules in Visual Studio .NET 2003 projects and debug the Fortran code when you debug other portions of your project.

Before debugging, you may need to include Fortran modules or libraries in projects that contain C and C++ source files. Although Microsoft does not provide a Fortran compiler with Visual Studio .NET 2003, you can include Fortran modules in Visual Studio .NET 2003 projects. To do this, compile the Fortran library or Fortran module with an option that produces a program debug database. Visual Fortran provides this option with the **-Zi** compiler option. This is similar to the process you use to create a program debug database for C and C++ programs. This creates a file with the .pdb extension that contains the debug symbols.

After you have created a debug version of your Fortran module or library, you can include it in a C or C++ project. The easiest way to do this is to add the debug version of the Fortran project as input for the Visual Studio linker.

To add the Fortran project

1. On the **Project** menu, click **Properties**.
2. In the **Configuration** list, click **Debug**.
3. In the **Platform** list, click **Win32**.
4. Click the **Linker** tab.
5. In the **Input** box, type the name of the Fortran object module or library you want to include.

Alternatively, you can use the Intel Visual Fortran compiler within Visual Studio .NET 2003 to develop Fortran applications, including static library (.lib), dynamic-link library (.dll), and main executable (.exe) applications. You can build your source code into several types of programs and libraries, either using Visual Studio .NET 2003 or working from the command line.

Note Use the IDE to build applications for IA-32 Windows-based systems only.

For building a Fortran project using Visual Studio .NET 2003, you need some additional settings. The following procedures describe the settings required in Visual Studio .NET 2003.

You must enter a search path so that the linker can find the module or library you added. The path must include the current working directory and the directories specified in the **Options** dialog box.

You can set the path and library and include directories for your Intel Fortran project environment on Visual Studio .NET 2003.

To add a new folder to the search path

1. On the **Tools** menu, click **Options**.
2. Click **Intel Fortran** in the list on the left of the dialog box.
3. In the **General** category, specify the directories in which the Visual Studio project system should look for files, as follows:
    - **Executables**. Specify the directories to be searched for executable files. (Works like the PATH environment variable.)
    - **Libraries**. Specify the directories to be searched for libraries. (Works like the LIB environment variable.)
    - **Includes**. Specify the directories to be searched for include files. (Works like the INCLUDE environment variable.)

Visual Studio .NET 2003 does not provide an option to specify the path for the .pdb file for your Fortran module. Therefore, you must place the .pdb file in the same directory as the library or object module you want to include.

You can also add a Fortran module or library directly to the project.

To add a Fortran object module or library directly

1. On the **Project** menu, click **Add New Item**. The **Add New Item** dialog box appears.
2. In the **Templates** tab, select **.obj** or **.lib**, depending on whether you need to insert an object module (.obj) or library (.lib).
3. Enter the file and the path for the Fortran object module or the library you want to add from the **Location** box or the browser.

Now you are ready to start debugging. You can start the debugger and step through code in either the C or C++ source or the Fortran source.

You can also debug the binaries created with the **Zi** compiler option using other Microsoft debug tools, such as WinDbg.

You may need to include a Fortran object module or library in a Visual Studio .NET 2003 project where a debug version of the object module or library is not available, or you may not need to include the Fortran routines in your debug session. If you do not need to debug the Fortran modules in a Visual Studio .NET 2003 project, you can include the release versions of either the object module or library as part of the link. You can continue to debug the C or C++ code. However, the debugger will not step into the Fortran source when a Fortran routine is entered. To accomplish this, include the release version of the Fortran object module or library in the **All Configurations** section of the **Project Settings** dialog box.

If you must include Fortran code as part of your debugging, remember to include the release versions of the Fortran object module or library in the **Win32 Release** section of the **Project Setting** dialog box.

Summary of Fortran Code Migration

The primary tasks in a Fortran code migration are focused on integration with libraries and the development environment. The actual Fortran code can usually be migrated or ported with little or no changes, provided that the level of Fortran compiler on the source platform is the same as the target platform. For example, if your application was developed on the source platform using Fortran 90, the target platform also needs a Fortran 90 compiler or Fortran 90 compatibility mode.

Issues related to source files and source control migration from UNIX to Windows are similar for Fortran because they are for C and C++. These issues are covered in Chapter 4, “Planning Phase: Setting Up the Development and Test Environments” of Volume 1: *Plan* of this guide.

Chapter 8: Developing Phase: Deployment Considerations and Testing Activities

This chapter discusses the key aspects of deploying and testing a migrated application on Microsoft® Windows® operating systems.

You can use the information provided in this chapter to identify the implementation requirements, such as environment variables, database connectivity, and migration of scripts, for creating the migrated environment. You will also be able to identify the deployment requirements, such as packaging and deploying of tools and administering the deployed Microsoft Win32® applications. This chapter also discusses various testing activities that you need to carry out in the Developing Phase.

Deployment Considerations

To ensure smooth deployment in the Deploying Phase, you need to address the following topics in the Developing Phase:

- Process environment
- Migration of scripts
- Database connectivity
- Building the application
- Deployment
- Configuration
- Packaging tools and installation
- Deploying applications
- Managing applications

The process for deploying the migrated application is discussed in detail in Volume 5, *Deploy-Operate* of this guide.

Process Environment

The process environment includes several key elements, which are explained in this section. The notable differences between these elements in Windows are also described briefly. This section discusses the Portable Operating System Interface (POSIX) environment in general because the deployment environment varies with respect to vendor and version of UNIX. This section provides you with the necessary information to set up or retrieve various environment-specific details in the UNIX and Windows environments.

Environment Variables

Every process has an environment block associated with it. An environment block is a block of memory allocated within the address space of the process. Each block contains a set of name value pairs. Both UNIX and Windows support process environment blocks. The particular differences may vary depending on which supplier and version of UNIX you are dealing with.

Note For information on conducting this comparison, refer to the MSDN article, "Changing Environment Variables," at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/changing_environment_variables.asp.

A summary of the notable differences between environment variables in Windows and POSIX is also provided at this URL.

Differences Between UNIX and Windows Environment Variables

Windows supports an ANSI version of the environment functions as well as a Unicode variant. The Unicode variant is preceded with a **_w** prefix. Using the **_w** prefix in your application helps ensure that the application is linked with the correct variant when compiled with _UNICODE or _MBCS preprocessor strings. In addition to the ANSI functions **putenv** and **getenv**, the Windows application programming interface (API) also supports the **GetEnvironmentVariable**, **GetEnvironmentStrings**, **ExpandEnvironmentStrings**, and **SetEnvironmentVariable** functions.

The following is a simple example of accessing the environment block. This example works equally well in both the UNIX and Windows APIs. This example shows only the ANSI functions. Using the ANSI functions provides you with the simplest method of converting your code from UNIX to the Windows API.

UNIX/Windows example: Accessing the environment block

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char *var, *value;

  if(argc == 1 || argc > 3) {
    fprintf(stderr,"usage: environ var [value]\n");
    exit(1);
  }
  var = argv[1];
  value = getenv(var);
  if(value)
    printf("Variable %s has value %s\n", var, value);
  else
    printf("Variable %s has no value\n", var);

  if(argc == 3) {
    char *string;
    value = argv[2];
    string = (char *)malloc(strlen(var)+strlen(value)+2);
    if(!string) {
      fprintf(stderr,"out of memory\n");
      exit(1);
    }
    strcpy(string,var);
    strcat(string,"=");
    strcat(string,value);
    printf("Calling putenv with: %s\n",string);
    if(putenv(string) != 0) {
      fprintf(stderr,"putenv failed\n");
      free(string);
      exit(1);
    }

    value = getenv(var);
    if(value)
      printf("New value of %s is %s\n", var, value);
    else
      printf("New value of %s is null??\n", var);
  }
  exit(0);
 }
```

(Source File: W_GetEnvVar-UAMV3C8.01.c)

Temporary Files

Both UNIX and Windows APIs support functions that create temporary files.

The **tmpnam()** function returns a pointer to a temporary file name. The **_tempname()** function does this as well, but you can also use it to specify the directory and file name prefix.

Computer Information

At times, it is necessary to obtain information about a computer. This is particularly important when an application is designed to support multiple users or different types of hardware and operating systems. Some of the pieces of information that applications require are as follows:

- Host name
- Operating system name
- Network name of the computer
- Release level of the operating system
- Version number of the operating system
- Hardware platform name

In UNIX, you use a combination of **gethostname** and **uname** functions to obtain this information. When using Windows, you have the option of using **gethostname**. However, **uname** is not available as standard in the Windows API. It is possible to add **uname** using a POSIX layer, which is possible by installing Windows Services for UNIX 3.5. Applications that use this function need to be rewritten to use a different set of services.

The Platform SDK has the functionality to obtain a set of information that is similar to that provided by the **uname** function. The Platform SDK mappings are covered in this text, but it is recommended that you consider using the Windows Management Instrumentation (WMI) API. The WMI interface is a superset to the Windows API for obtaining information about the computer. It is highly extensible and supports not only static information about a platform, but also dynamic information such as configuration and performance data. Another source to consider is the Active Directory Service Interfaces (ADSI), a COM interface that facilitates access to information stored in the Microsoft Active Directory® directory service database for the enterprise. Both these interfaces represent the preferred mechanism for gathering information about Windows Server™ 2003.

Note For a complete list of the system information functions provided by the Platform SDK, you can refer to "System Information Functions" in the online platform SDK documentation on the MSDN Web site at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sysinfo/base/system_information_functions.asp.

The two functions **GetVersionEx** and **VerifyVersionInfo** are used to get extended information about the operating system and to compare the operating system versions on Windows.

UNIX example: Using system information

```
#include <unistd.h>
#include <stdio.h>
#include <sys/utsname.h>

int main()
{
  char computer[256];
  struct utsname uts;

  if(gethostname(computer, 255) != 0 || uname(&uts) < 0) {
    fprintf(stderr, "Could not get host information\n");
    exit(1);
  }

  printf("Computer host name is %s\n", computer);
  printf("System is %s on %s hardware\n", uts.sysname, uts.machine);
  printf("Nodename is %s\n", uts.nodename);
  printf("Version is %s, %s\n", uts.release, uts.version);
  exit(0);
}
```

(Source File: U_SysInfo-UAMV3C8.01.c)

Win32 example: Using system information

```
#define _WIN32_WINNT 0X0500
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>

void errabt(char *msg)
{
  fprintf(stderr, msg); // use GetLastError for more detailed info.
  exit(1);
}

void main()
{
  DWORD nSize= 255;
  char computer[256];
  char nodename[256];
  SYSTEM_INFO siSysInfo;       // Struct for hardware info
  OSVERSIONINFO siVerInfo;  // Struct for version info

  GetSystemInfo(&siSysInfo); // Get hardware OEM

  // Get major and minor number
  ZeroMemory(&siVerInfo, sizeof(OSVERSIONINFO));
  siVerInfo.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
  if (!GetVersionEx((OSVERSIONINFO *) &siVerInfo))
    errabt("Could not get OS Version info\n");

  nSize = 255;
  if (GetComputerNameEx(ComputerNameNetBIOS, computer, &nSize) ==
FALSE)
    errabt("Could not get NETBIOS name of computer\n");

  nSize = 255;
  if (GetComputerNameEx(ComputerNameDnsFullyQualified, nodename,
&nSize) == FALSE)
    errabt("Could not get FQDNS Name of computer\n");

  printf("Computer host name is %s\n", computer);
  printf("System is %u on %u hardware\n",
         siVerInfo.dwMajorVersion,siSysInfo.dwProcessorType);
printf("Nodename is %s\n", nodename);
  printf("Version is %d.%d %s (Build %d)\n",
    siVerInfo.dwMajorVersion,
    siVerInfo.dwMinorVersion,
    siVerInfo.szCSDVersion,
    siVerInfo.dwBuildNumber & 0xFFFF);
  exit(0);
}
```

(Source File: W_SysInfo-UAMV3C8.01.c)

Logging System Messages

Logging diagnostic messages in UNIX is carried out by writing formatted output to the system logger. The message is written to system log files, such as USERS, or forwarded to the appropriate computer. If a log daemon process is not running, the log information may be written to a standard log file such as /var/adm/log/logger.

The daemon syslogd in UNIX contains numerous levels of logged information, as listed in Table 8.1.

Table 8.1. UNIX Logging System Messages

Priority	Description
LOG_EMERG	A panic condition.
LOG_ALERT	A condition that should be corrected immediately.
LOG_CRIT	Critical conditions such as hard device errors.
LOG_ERR	Errors.
LOG_WARNING	Warnings.
LOG_NOTICE	Non-error–related conditions.
LOG_INFO	Informational messages.
LOG_DEBUG	Messages intended for debug purposes.

In contrast, the Windows event log supports logging levels, as listed in Table 8.2.

Table 8.2. Windows Event Logging Messages

Priority	Description
EVENTLOG_SUCCESS	Information events indicate infrequent but significant successful operations. For example, when Microsoft SQL Server™ successfully loads, it may be appropriate to log an information event stating that "SQL Server has started." Note that while this is appropriate behavior for major server services, it is generally inappropriate for a desktop application (for example, Microsoft Excel®) to log an event each time it starts.
EVENTLOG_ERROR_TYPE	Error events indicate significant problems that the user should know about. Error events usually indicate loss of functionality or data. For example, if a service cannot be loaded as the system starts, it can log an error event.
EVENTLOG_WARNING_TYPE	Warning events indicate problems that are not immediately significant, but may indicate conditions that can cause problems in the future. Resource consumption is a good candidate for a warning event. For example, an application can log a warning event if the disk space is low. If an application can recover from an event without loss of functionality or data, it will generally classify the event as a warning event.
EVENTLOG_INFORMATION_TYPE	Information events indicate infrequent but significant successful operations. For example, when SQL Server successfully loads, it may be appropriate to log an information event stating that "SQL Server has started." Note that while this is appropriate behavior for major

Priority	Description
	server services, it is generally inappropriate for a desktop application (for example, Excel) to log an event each time it starts.
EVENTLOG_AUDIT_SUCCESS	Success audit events are security events that occur when an audited access attempt is successful. For example, a successful logon attempt is a successful audit event.
EVENTLOG_AUDIT_FAILURE	Failure audit events are security events that occur when an audited access attempt fails. For example, a failed attempt to open a file is a failure audit event.

As you can see, the Windows event logging mechanism supports a smaller selection of event priorities than UNIX. You can augment the priority status of event messages by including category information and binary data in the event log. This additional event information is part of the Windows example (following the UNIX example).

UNIX example: System logging

```
#include <syslog.h>
#include <syslog.h>
#include <stdio.h>

int main()
{
  FILE *fp;

  fp = fopen("Bad_File_Name","r");
  if(!fp)
    syslog(LOG_INFO|LOG_USER,"error - %m\n");
  exit(0);
}
```

(Source File: U_SysLog-UAMV3C8.01.c)

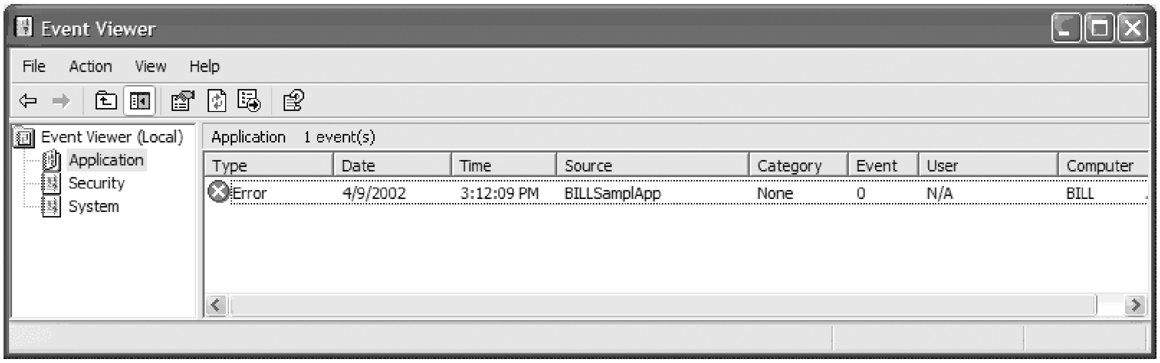
On a typically configured Linux system, this message would be logged to /var/log/messages, and on a Solaris system, the message would be logged to /var/adm/messages. For more specific information, consult the /etc/syslog.conf file. Specifically, a *.info entry will specify the file where the message is going to be logged.

Windows example: System logging

```
#include <windows.h>
#include <stdlib.h>

void main()
{
  HANDLE h;
  LPSTR mstr = "This is an error from my sample app.";
  h = RegisterEventSource(NULL, // uses local computer
       TEXT("BILLSamplApp"));   // source name
  if (h == NULL)
    exit(1);

  ReportEvent(h,        // event log handle
      EVENTLOG_ERROR_TYPE, // event type
      0,          // category zero
      0,          // event identifier
      NULL,         // no user security identifier
      1,          // one substitution string
      0,          // no data
            (LPCSTR*)&mstr,    // pointer to string array
      NULL);        // pointer to data

  DeregisterEventSource(h);
      exit(0);
}
```

(Source File: W_SysLog-UAMV3C8.01.c)

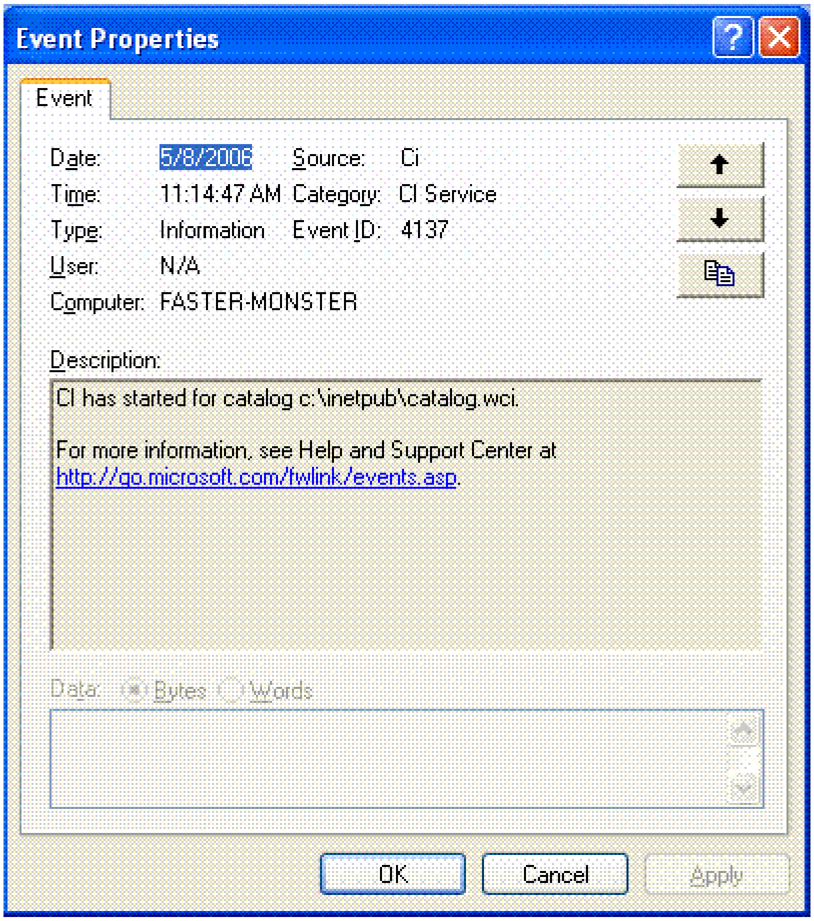
In the preceding example, the source name to the **RegisterEventSource** call is not available in the system registry. As a result, you will not see valid mapping or lookup data when you view the event log with the Event Viewer. After running this code, Eventvwr.exe would display a window as shown in Figure 8.1.



Type	Date	Time	Source	Category	Event	User	Computer
Error	4/9/2002	3:12:09 PM	BILLSamplApp	None	0	N/A	BILL

Figure 8.1. Windows Event Viewer

Double-clicking the error line opens a detailed view of the event (depicted in Figure 8.2).



Date: 5/8/2006 Source: Ci
Time: 11:14:47 AM Category: CI Service
Type: Information Event ID: 4137
User: N/A
Computer: FASTER-MONSTER

Description:

CI has started for catalog c:\inetpub\catalog.wci.

For more information, see Help and Support Center at http://go.microsoft.com/fwlink/events.asp.

Figure 8.2. Details of an event in Windows Event Viewer

The preceding example is a very simple example of generating log information and posting it to the Windows event log. A complete application would use more of the Platform SDK facilities to create an application entry in the registry or perhaps create an entirely separate event log file.

Note For a complete discussion of the details and complexities of event logging in Windows, refer to “Set event logging options” on the TechNet Web site at http://technet2.microsoft.com/WindowsServer/en/Library/0473658c-693d-4a06-b95b-ebe8a76648a91033.mspx.

Migrating Scripts

This section describes the process of porting UNIX shell scripts to the Windows environment. Following are the steps involved in the porting process:

1. Evaluating the script migration tasks.
2. Planning for fundamental platform differences.
3. Considering the target environments.

The steps in the process are described in more detail later in this section. This section helps you choose the appropriate porting approach and the target scripting language in the Windows environment.

Scripts fall into the following two basic categories:

- Shell scripts, such as Korn and C shell.
- Scripting language scripts, such as Perl, Tcl, and Python.

Shell and scripting language scripts tend to be more portable than compiled languages, such as C and C++. A scripting language such as Perl is compatible with most platform features. However, the original developer might have used easier or faster platform-specific features, or just might not have taken cross-platform compatibility into consideration.

The choice of porting approach depends on the source script type and whether the target environment is Windows only, Windows plus Interix, or uses CGI scripts.

In the Windows-only environment, a solution is to write all common scripts in Perl because there are several versions of Perl available. If software is to be maintained on UNIX and Windows-based systems, writing all-new scripts in Perl, and even converting some existing shell scripts to Perl, is a good strategy.

Evaluating the Script Migration Tasks

Before script migration begins, all required tasks need to be considered. To identify script migration tasks, consider the following questions:

- What are the scripting languages being used?
- Does the script rely on the syntax of the shell?
- Does the script use substantial external programs?
- Does the script use any platform-specific services?
- Does the script use extensions that rely on third-party libraries?
- Does the script use or rely on nonportable concepts for essential functionality?
- Can a quick port be done now, with a rewrite later?
- Does the developer understand enough of the original code to quickly locate the issues and then make the changes necessary to port to a new platform?

By answering these questions, script migration tasks can be evaluated and defined. Redesigning and rewriting portions of the application might be easier than porting because it is more efficient to take advantage of native features.

Planning for Fundamental Platform Differences

While porting scripts, the code must address some inevitable fundamental differences between the platforms. The following areas, which are described in more detail in later sections, are often sources of script migration issues:

- File system interaction
- Environment variables
- Shell and console handling
- Process and thread execution
- Device and network programming
- User interfaces (UIs)

File System Interaction

UNIX and Windows-based systems interact differently with the file system. The UNIX path separator is a forward slash (/), whereas Windows uses the backslash (\). The root of UNIX files is represented by the forward slash (/), but Windows uses locally mounted drives ([**A-Z**]**:**) and network-accessible drives using the Universal Naming Convention (**\\***ServerName******SharePoint******Dir*****).

The first things you should correct in any code to be migrated are hard-coded file paths. These paths are commonly used to find initialization or configuration files (that is, to set up environment variables or application paths). One common mistake during the initial porting work is to refer to a Windows-based file in the native form. The problem is that the backslash (****) is also the common escape character. As a result, the path C:\dir\text.txt is translated as C:dir ext.txt. (The space is a single tab character.)

In most cases, Windows can handle the forward slash (/) as a path separator. However, when building cross-platform paths, scripting language compilers can misinterpret even correctly used file path separators or methods.

Unlike UNIX file systems, Win32 file systems are not case-sensitive. They may preserve the case of file names, but the same directory cannot contain two different files where only the case of the file name letters differs (for example, file.txt and FILE.txt). Windows also does not allow users to create a file with the same name as the directory in which it is created.

Note When hard coding paths in a script, certain Windows directories naming changes depend on the native language. For example, the directory named C:\Program Files\ in the English version of Windows is named C:\Programme\ in the German version.

The exact names for paths and other information that may be critical in porting your code are often found in the Windows registry. For example, the correct path for the Program Files directory can be found in HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\ProgramFilesDir.

Windows registry is a central database of information about your Windows system. Windows registry contains information such as what hardware is present on the system, how the hardware and system are configured, and what applications are installed on the system. The registry provides fine-grained security. Each registry key can be protected with an access control list (ACL) in exactly the same way that files can be protected.

You can refer to the registry when other platform-independent methods are not available. Use the **regedit** command to peruse the Windows registry. Some of the information stored in the registry is also available by using language APIs, which are safer to use.

Environment Variables

Both Windows and UNIX use environment variables. Although Windows maintains an environment array, its contents are not similar to UNIX. The Windows environment array is not case-sensitive, so the environment variables *PATH*, *path*, and *PaTh* all refer to the same item. The PATH variable is similar in purpose across platforms (for example, shells search the directories specified in the *PATH* environment variable for executables and scripts), but Windows uses a semicolon (**;**) as a separator, whereas UNIX uses a colon (**:**). Fortunately, compiled languages usually have features that handle the differences in usage of the *PATH* variable.

Commonly used UNIX environment variables are *HOME, PATH, USER*, and *TEMP*. Windows also has the *PATH* and *TEMP* variables. To determine which environment variables are used in a Windows installation, use the abstractions in a compiled language or look in the Windows registry. As an alternative, you can use the following technique.

To see the full contents of the environment

1. Right-click **My Computer**, and then click **Properties**.
2. In the **System Properties** dialog box, click the **Advanced** tab.
3. Under Environment Variables, click **Environment Variables**.
4. In the **Environment Variables** dialog box, view and modify the environment.

Note that Windows has separate user and system environments. Administrator rights are required to modify the system environment.

Scripts commonly require a temporary data file, which is usually hard-coded to reside in /tmp on UNIX. On Windows and UNIX, use the TEMP environment variable instead to refer to an acceptable temporary file directory. Some scripts also rely on environment variables beginning with LC_, which indicates the locale information for that system.

Files are not always the same at the binary level. For example, Windows uses CRLF (carriage return/linefeed or characters \015\012) at the end of a line, whereas UNIX uses only LF. Script environments provide methods for handling this transparently. Another discrepancy is that **^Z** (character \032) represents the end-of-file character. A UNIX script with this character embedded in code might ignore it, and Windows might stop reading the file at that point.

Shell and Console Handling

The shell is found on all UNIX desktops. Windows provides a command shell.
Windows Server 2003 stores the path to the shell in the COMSPEC variable of the environment array. Developers interact with the command shell during testing, but it can interfere or act in unexpected ways during the ordinary operation of a script. Some languages can run without any attachment or specific connection to a terminal. For guidance on how to make the console behave as required, refer to the language specifics.

Some scripts call the shell to reuse existing commands, such as **cat**, **ls**, **sendmail**, **date**, and **grep**. Relying on the shell is not recommended because it not only reduces processing power by creating external process execution overhead, it is also highly nonportable. To avoid portability issues, it is better to rely on the methods that the language provides.

For example, the following example might not be portable:

```
set date [exec date “+%D %H:%M”]
```

The following example is portable:

```
set date [clock format [clock sec’s] -format “%m/%d/%y %H:%M”]
```

Note that invoking commands from the shell automatically use wildcard expansion (usually referred to as *globbing*). When the script relies on globbing, you should use the language methods for file globbing to expand file names. Where it is unavoidable to call the shell, it is important to note that the Windows command shell has different native commands and quoting rules.

Process and Thread Execution

The script might need to deal with process manipulation, especially if external system calls are unavoidable. In a language that supports process manipulation, the features are usually portable to Windows Server 2003. However, it is still necessary to evaluate all uses of process manipulation to ensure that the application code is manipulating the correct Windows processes.

It is common in UNIX to manage processes by passing signals, especially for daemon processes and system administration tasks. Signal handling, when handled by the language, is similar to process manipulation. Some uses of signal handling are portable from UNIX to Windows Server 2003, but not all signals are relevant. Windows uses an event passing model. A UNIX daemon process ported to Windows needs to respond to these events. When porting a UNIX daemon on Windows, it is necessary to create a Windows service that provides essentially the same functionality.

It is important to note that a **fork** command can have a different behavior in UNIX, depending on the language. If the **fork** command is used in UNIX, it is highly recommended that you look at alternative techniques for achieving the same result on Windows. The best solution is to switch to using threads.

Device and Network Programming

Many applications built today use a client/server model or must follow network or interprocess communication (IPC) protocols, such as HTTP, TCP/IP, and UDP. Scripting languages provide varying levels of abstraction over the standard system mechanism for communicating with files and sockets. Because some are more portable than others, it is important to examine socket handling when porting code. Methods for IPC outside socket programming or communicating through a pipe should be avoided because they are normally nonportable. A well-known remote procedure call (RPC) mechanism that works well across platforms and fits well into Web server applications is simple object access protocol (SOAP), which most scripting languages already support.

An application that communicates with the serial port or other system device can use the same protocol for interacting with the device, but must often address the device differently. For example, a serial device on UNIX can be addressed as the special file /dev/ttya. On Windows, it is addressed as COM1.

User Interfaces (UIs)

Many scripting languages have access to one or more graphical user interface (GUI) toolkits. If the language used in script has a GUI toolkit, it is important to determine the portability of that toolkit across platforms.

Tk is a GUI toolkit common to Tcl, Perl, and Python. It is fully cross-platform compatible between UNIX and Windows. Some of the finer points of cursor and font handling can vary between these systems because of the underlying operating system differences.

Scripting Environment

The Common Gateway Interface (CGI) protocol is the standard interface used by Web servers to run programs and scripts that handle dynamic content. Usually, CGI portability is not an issue because CGI is a standardized interface available under all major Web servers. CGI is falling out of favor and is being replaced by other techniques that cost the operating system less and scale better. Any language can be used as a CGI language if it supports reading and writing STDOUT and STDIN console handles, and chances are that many existing scripts are CGI-based. In recent years, many Web server plug-ins have been written for scripting languages to work around performance limitations in CGI, although using these plug-ins sometimes requires minor changes to the CGI script itself. Apache has direct language plug-ins for Perl (mod_perl), PHP (mod_php), and Tcl (mod_tcl). Through the Internet Server API (ISAPI), Internet Information Server (IIS) has a direct language plug-in for Perl called PerlEx.

Database Connectivity

This section describes various database connectivity mechanisms compatible with UNIX and Windows applications and provides an overview of each of these mechanisms.

Microsoft offers many data access technologies for various database management systems (DBMSs). Microsoft Data Access Components (MDAC) includes ActiveX® Data Objects (ADO), OLE DB, and Open Database Connectivity (ODBC). Data-driven applications can use these components to easily integrate information from a variety of sources—both relational (SQL) and nonrelational.

Note Detailed information on MDAC is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/mdacsdk/htm/dasdk_overview.asp.

As stated, MDAC includes:

- **ADO**. ADO provides consistent, high-performance access to data and supports a variety of development needs, including the creation of front-end database clients and middle-tier business objects that use applications, tools, languages, or Internet browsers.

  ADO provides an easy-to-use interface to the OLE DB, which provides the underlying access to data. It uses the COM automation interface available from all leading rapid application development (RAD) tools, database tools, and languages.

  Note Additional information is available at

  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ado270/htm/dasdkadooverview.asp.

- **OLE DB**. Microsoft OLE DB is a set of COM-based interfaces that expose data from a variety of relational and nonrelational data providers. OLE DB interfaces provide applications with uniform access to data stored in diverse information sources.

  OLE DB comprises a programmatic model consisting of:

  - **Data providers**. These contain and expose data.
  - **Data consumers**. These use data.
  - **Service components**. These process and transport data (such as query processors and cursor engines).

  In addition, OLE DB includes a bridge to ODBC to enable continued support for the broad range of ODBC relational database drivers.

  The following OLE DB providers are available:

  - OLE DB provider for ODBC
  - OLE DB provider for Oracle
  - OLE DB provider for SQL Server

  Note Additional information on these OLEDB providers is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbprovmicrosoft_ole_db_providers_overview.asp.

  The Microsoft OLEDB core services and the Microsoft SQL Server OLEDB provider support 64-bit Windows.

  Note For more information on this, refer to MSDN Web site at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/mdacsdk_oledb_64bit.asp.

- **ODBC**. ODBC is a C programming language interface that makes it possible for applications to access data from a variety of DBMSs. Using ODBC, an application can access data in diverse DBMSs through a single interface. The application is independent of any DBMS from which it accesses data. Users of the application can add software components called drivers, which create an interface between an application and a specific DBMS.

  ODBC drivers provide access to the following types of data sources:

  - Microsoft Access
  - Microsoft Excel
  - Paradox
  - DBASE
  - Text
  - Oracle
  - Visual FoxPro®

  Note Additional information on ODBC drivers is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/odbc/htm/odbcodbc_drivers_overview.asp.

  The ODBC headers and libraries shipped with MDAC 2.7 SDK allow programmers to write code for the new 64-bit platforms. An application with code that uses the ODBC defined types in the ODBC libraries of MDAC 2.7 can use the same source code both for 64-bit and 32-bit platforms.

  Note Additional information on this is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/odbc/htm/dasdkodbcoverview_64bit.asp.

Building the Application

This section describes the Visual Studio® .NET 2003 integrated development environment (IDE) that can be used to build and debug Windows applications. IDEs typically provide all development tools needed for programming, including compilers, linkers, and project/configuration files that generate complete applications, create new classes, and integrate those classes into the current project. IDEs also include file management for sources, headers, documentation, and other material to be included in the project. IDEs can also include the creation of UI elements, resources like icons, bitmaps and cursors, and marinating the language resources like string tables. Other typical capabilities include the debugging of the application and the inclusion of any other program needed for development by adding it to a **Tools** menu.

Visual Studio .NET 2003 includes a complete set of development tools for building reusable Win32/Win64 applications. With Visual Studio .NET 2003, you can:

- Do programming through wizards, perform drag-and-drop editing, and reuse program components from any of the Visual Studio .NET languages. Because of the use of programming wizards, you need to write less code.
- Write code more quickly by minimizing errors with syntax and programming assistance within the editor.
- Integrate dynamic HTML, script, and components into Web solutions.
- Manage Web sites from testing to production by means of integrated site management tools.
- Create and debug Active Server Pages (ASP).
- Use design-time controls to visually assemble data-driven Web applications.

Visual Studio .NET 2003 also includes the *Windows 2000 Developer's Readiness Kit*, which contains developer training and technical resources.

Note Additional information on using Visual Studio .NET is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vxconATourOfVisualStudio.asp.

Deployment

The following sections discuss how you can configure, package and install, and deploy your application, as well as manage it. It specifically talks about using the Microsoft Windows Installer service and other tools for packaging your application. You can use the information provided in this section to identify critical parameters necessary for deployment, such as the best packaging tool and the most suitable mechanism for deploying your Win32/Win64 application.

Configuration

Occasionally, it is desirable to store information on a user's computer. The reason for doing so may be to store information about program settings, which should persist from one invocation of the program to the next, or about registration details, or the connection string for the database.

The registry is a system-defined database in which applications and system components store and retrieve configuration data. The data stored in the registry varies according to the version of Windows. Applications use the registry API to retrieve, modify, or delete registry data.

Note Additional information on the registry API is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sysinfo/base/registry_reference.asp.

Packaging Tools and Installation

This section explains how to package your migrated Win32/Win64 application and install it into a Windows environment. The standard method of packaging applications in a Win32/Win64 environment is to use the Windows Installer service. This section covers the Windows Installer and some of the tools that you can use to package your application.

Windows Installer Service

The Windows Installer service uses the Windows Installer package, which is now the standard way that application developers deliver software for the Windows platform. Using the format and a common set of actions standardized by Microsoft, tool vendors can add value in creating, editing, and distributing customized Windows Installer packages supporting such features as self-repair, rollback of the installation, and installation of the selective features of the application.

The Windows Installer package packages the libraries that are implicitly linked in the application. But if the application uses explicit linking of libraries, these libraries should be packaged by adding them explicitly to the packaged application.

The .msi file format contains all of the instructions that a program needs to install itself, which includes locations of files, movements or deletions of existing files, creation of shortcut icons, a **Start** menu entry, registry settings, ACL changes, Windows service installation or changes, and COM component registration. The program files may be contained in the .msi or in one or more .cab (compressed) files. The .msi uses database tables to describe the features and components of the product, the relations between the two, and all of the actions required to install, upgrade, or uninstall the application.

The installer will copy the files of the application to their correct locations from an embedded file or from the accompanying .cab file or files. Usually, the Windows Installer is invoked when a computer starts and logs on to the domain (by using an Active Directory computer account) or when a user logs on to the computer. The Group Policy feature of Active Directory is used to attach .msi installation packages to these events, although you can also invoke the installer locally by using other means such as scripting or the scheduler service.

The installation process leaves a copy of the .msi instructions on the local computer. Each time an application is launched, it checks the feature and component listings in the local .msi to see if everything is still intact. Missing or corrupt components can trigger an automatic repair known as *self-healing*. Self-healing can also be triggered during the operation of an application when a component is dynamically loaded. This feature is useful when .msi files are well architected by the software developer, but it is often turned off by administrators when the .msi has been repackaged. This is because many large applications are, by necessity, built as single feature packages, and self-healing would force the administrator to reinstall the entire application.

Installation on Demand

Install-on-Demand is one of the most useful features of the Windows Installer service. Install-on-Demand is the feature that prompts for the location of the installation media after selecting an item that you have not used before when using a product such as Microsoft Office. By using Install-on-Demand, you can leave components that you do not access frequently (grouped into feature sets) uninstalled and on the server until a user invokes them. You can apply this to all applications in the case of Group Policy software advertisements, where only a desktop shortcut is deployed initially, or you can apply it to application features such as spelling checkers, graphic libraries, or charting. To use this feature properly, you must understand how to divide the application into independent chunks (or components), and you must then choose which components to share across an entire installation and which components to hold back until they are needed. The local .msi database maintains a list of installation points where it can find the necessary files at the time of installation. You can edit this list by using subsequent update (.msp) files to update network locations, such as Distributed File System (DFS) shares.

Installation Rollback

As the installation occurs, the service backs up overwritten files and keeps track of any changes that are made to the system, such as registry entries or ACL changes. If the setup is not run to completion, an automatic rollback restores the original state of the system. You can run Windows Installer with verbose logging options to record these events. However, there are no built-in alert mechanisms in Group Policy. Most large enterprises using Active Directory as their only means of distributing software have developed scripts to query the .msi log files or interrogate the .msi local database by using Windows Management Instrumentation (WMI).

Installation Auditing

The following Visual Basic Scripting Edition (VBScript) code interrogates the local .msi database and lists installed packages. You can use this, in conjunction with remote scripting, as the basis of a simple installation audit.

Windows example: Installation auditing

```
Dim installer, product
Dim version
Dim productList, productString

productList = ""

Set installer = Wscript.CreateObject("WindowsInstaller.Installer")

For Each product In installer.Products
version = CLng(installer.productInfo(product, "Version"))
version = (version\65536\256) & "." & _
(version\65535 Mod 256) & "." & _
(version Mod 65536)

productString =installer.productInfo(product, "ProductName")_
& vbCrLf & " ID: " & product _
& " Version: " & version & vbCrLf

productList = productList & productString & vbCrLf
Next

If productList <> "" Then
productList = "Found " & installer.products.Count & _
" applications" & vbCrLf & vbCrLf & productList
Else
productList = "No .msi applications listed."
End If

WScript.Echo productList
```

The following is an example of output from this script:

```
E:\>cscript ListMSIDB.vbs
Microsoft (R) Windows Script Host Version 5.1 for Windows
Copyright (C) Microsoft Corporation 1996-1999. All rights reserved.

Found 7 applications

Windows Server 2003 Administration Tools
 ID: {B7298620-EAC6-11D1-8F87-0060082EA63E} Version: 5.0.0
Microsoft Windows Services for UNIX
 ID: {E8A81EF0-40DB-4B5B-ABE8-558D69CE2F09} Version: 7.0.1620
Hummingbird Exceed
 ID: {CFBD3858-2164-42B0-84A2-576C18C85082} Version: 7.1.0
Microsoft Office XP Professional with FrontPage
 ID: {90280409-6000-11D3-8CFE-0050048383C9} Version: 10.0.2627
WebFldrs
 ID: {6F716D8C-398F-11D3-85E1-005004838609} Version: 9.0.3501
Windows Server 2003 Support Tools
 ID: {242365CD-80F2-11D2-989A-00C04F7978A9} Version: 5.0.2072
Microsoft Windows Server 2003 Resource Kit
 ID: {4E1F3FCF-B205-427F-B52B-D13BDFB6526C} Version: 5.0.2092
```

Security Rights and the Windows Installer Service

The Windows Installer service, when invoked by Active Directory Group Policy, runs a managed installation. This process runs under the Local System account, which has administrative rights. This allows applications to be installed on systems that are locked down (that is, systems on which the end users have limited rights and abilities).

Update Files

Another aspect of the Windows Installer service is the .msp, or update file. This is a specially formatted .msi that can identify and update existing installations of itself by unique product and version numbering. Many organizations will create build images by using the .msi format so that future updatees and upgrades can be deployed by using Group Policy in Active Directory.

Window Installer Service Transforms

If you have worked with Microsoft Office 2003 or Microsoft Office XP and are familiar with the Resource Kit Custom Installation Wizard, you have seen the transform technology of the Windows Installer. Transforms allow you to amend the installation instructions in the .msi file on the fly. These are usually authored using one of the commercial editing tools such as WinInstall or InstallShield.

A limited variety of UI widgets are available to the Windows Installer to gather user input during the installation and to look up information in outside files.

Creating New Windows Installer Service Packages

There are many tools available for packaging, distributing, installing, and managing applications in the Windows Installer format (that is, .msi files). Microsoft Visual Studio .NET 2003 can create installation packages in the .msi format. In addition, there are third-party tools that help you create and manage Windows Installer packages. These products typically provide the following features:

- An IDE for developing installation packages.
- Installation script editors.
- Installation debuggers.
- Options for Internet-based installations.
- Support for password and digital signature security options on installation packages.
- Support for the Windows Installer update files (.msp files).
- Support for the Windows Installer transforms file (.mst files).
- Source control integration.

The following are three third-party products that you can also use:

- InstallShield Developer

  Note Information is available at http://www.installshield.com.
- Wise for Windows Installer

  Note Information is available at http://www.wisesolutions.com.
- Veritas WinINSTALL

  Note Information is available at http://www.veritas.com.

Repackaging Applications

If a software installation process that has already been created does not support the Windows Installer (.msi) standard, you can use a repackaging application. At a minimum, a repackaging application will allow you to:

- Create fully featured Windows Installer setups by capturing installations that are not based on Windows Installer.
- Allow installations to be customized.
- Check and resolve any installation conflicts.

The following are two repackaging applications that you can use:

- InstallShield AdminStudio.

  Note Information is available at http://www.installshield.com.
- Wise Package Studio

  Note Information is available at http://www.wisesolutions.com.

Deploying Applications

The following subsections describe major activities during the deployment and the various policies used during the deployment process.

Deploying Applications with Group Policy Objects

Active Directory supports a technology known as Group Policy. You can assign Group Policy objects (GPOs) to users or computers, and you can associate them with any of the hierarchical containers that make up the directory structure. This means, for instance, that you can apply a policy to all the computers in an engineering department at a particular site or even across the organization, while the computers in an accounting department have their own policies. GPOs are filtered by the user groups in Active Directory so that you can keep precise control over applications of the users.

GPOs can set and enforce hundreds of settings on desktop computers, including all of the security settings, but the setting applicable here is the software distribution policy setting. You use the software distribution policy to deploy Windows Installer files (.msi files). The Windows Installer service, **msiexec.exe**, can be set by Group Policy to run with elevated (administrator-level) privileges. Thus an installation program that needs access to resources that a typical user would not have access to (for example, directories and registry entries) can still operate without the user having power user or administrator privileges.

Deploying Applications with Systems Management Server

Applications can be deployed using the Group Policy software distribution feature of Active Directory. However, there are several limitations of using GPOs for software deployment. These are addressed by Microsoft Systems Management Server (SMS). Here is a summary of the main reasons for using SMS instead of GPOs for application deployment:

- Active Directory Group Policy requires the applications to be in the Windows Installer (.msi) format, whereas SMS can deploy any executable package, including setup programs, scripting, and batch files. Many large applications include legacy setup architectures that are difficult or impossible to replicate in a repackaged .msi installation.
- Group Policy requires a user to log on or a computer to be restarted to initiate a software deployment policy.
- SMS has extensive Microsoft SQL Server-based reporting capabilities.
- SMS includes an extensive hardware and software inventory.
- SMS allows you to query the client computer before installation to ensure adequate disk space, memory, operating system version, and other software dependencies.
- SMS does not require Active Directory, although SMS can use Active Directory if it is available.
- Software installations can be advertised to users through desktop shortcuts, the SMS client icon, and Control Panel. Software installations can also be pushed to a client without user intervention.
- SMS allows you to define computer groups separately from Active Directory users and groups, based on inventory information.

Deploying Win32/Win64 Applications

This section describes different methods of deploying Win32/Win64 applications and how you can use them.

Deploying Win32/Win64 Applications by Pushing Them to the Desktop

Active Directory Group Policy software deployment or other systems that rely on a user logging on or a computer being restarted might need to have a second method of delivery that can be deployed without user intervention to client desktops.

SMS and other enterprise-level systems can achieve this as part of their typical client-server interaction. Other methods of achieving this include remote scripting or maintaining a service (daemon) on each desktop that checks for updates periodically.

Two-Phase Deployment of Win32/Win64 Desktop Applications

When deploying locally installed applications, you might want to avoid the distribution of large installation images over the network over a short period of time. To do so, you can use a two-stage approach, sometimes referred to as a knife-edge installation. In this scenario, the two stages are as follows:

1. **Deploy the installation image**. Deploy an .msi or SMS package that is designed to do nothing more than copy a potentially large installation image to the local disk of each user, possibly in a partition reserved for this purpose.
2. **Schedule the installation job**. A second package or job is then scheduled with the actual installation instructions that operate against this local image. This can be an incremental deployment over several days or weeks.

In this way, large numbers of users can simultaneously install a new version of an application without affecting the network and without raising data compatibility issues.

Another benefit of this technique is that multiple versions of the installation image of the application could be stored on the local drive for rapid rollback or piloting new versions. To maintain this rolling cache of images, you need well-tested install and uninstall jobs, packages, and processes.

If a deployment system such as SMS is in place with some additional Wake-On-LAN support, you can deliver the image deployment and installation packages outside ordinary working hours.

Large package deployments can also use compression technologies such as WinZip to deploy the package without dependence on the .msi format.

Side-by-Side Deployment of Win32/Win64 Applications

Although the Windows platform has been a successful development platform in part because of its built-in component-sharing mechanisms, these same shared components have also caused administrative headaches. Components from Microsoft (which are provided as part of the base operating systems, option packs, service packs, and various add-ins) and numerous third-party sources save developers countless hours. However, true backward compatibility means that shared components must function exactly as they did in previous versions while providing new functionality. In the real world, this is difficult to achieve because all configurations in which the component may be used need to be tested.

The practical functionality of a component is also not easily defined. Applications may become dependent on unintended side effects that are not considered part of the core function of the component. For example, an application may become dependent on an anomaly in the component, which when fixed causes the application to fail. The fact that dynamic-link libraries (DLLs) have been upgraded to newer internal versions while keeping the same names has also caused confusion.

This lack of backward compatibility can result in the inability to deploy a new application without breaking applications that are already deployed or compromising the functionality of the new application. To provide for successful sharing while enhancing application stability, Microsoft introduced side-by-side sharing starting in Windows 98 Second Edition and in Windows Server 2003, creating a way to share components through isolation.

With side-by-side components, multiple versions of the same component can be installed, and applications can use the one version that is most suitable.

Two different processes can load different versions of a Win32/Win64 or COM component at the same time and, independently, unload those components as required.

As outlined in the Windows Server 2003 logo certification guidelines, the best practice is to develop new applications and components with side-by-side use in mind, but there is also a way to selectively isolate the majority of the existing components by using a Windows redirection mechanism. Although redirection does not require changing any code, it does need to be thoroughly tested to ensure that the applications on the system continue to operate normally. Because these components may now be distributed into the directories of many applications, there is also an increase in the complexity of administering the components.

Creating new components for side-by-side use is the best way to guarantee that applications can load them independently. In this case, the component must be developed with careful attention to where global data and state information are stored. Any other factors that can affect having multiple versions of components in memory simultaneously must also be addressed. For instance, instead of storing a particular setting using a registry key such as:

```
HKEY_CURRENT_USER\Software\Vendor\ComponentName\RegKeyName = SomeValue
```

the component would be better isolated using a version-specific key such as:

```
HKEY_CURRENT_USER\Software\Vendor\ComponentName\VersionNumber\RegKeyNam
e = SomeValue
```

or even a version- and application-specific key:

```
HKEY_CURRENT_USER\Software\Vendor\ComponentName\VersionNumber\
ApplicationSpecificName\RegKeyName = SomeValue
```

Shared memory structures such as memory-mapped files and named pipes also need to be taken into account and renamed or relocated on a per version basis. Better still, a component should be designed to be as stateless as possible and to let the client application handle state and user-specific data as much as possible. Where the component really does need to store its own state information, it should use a method or property of the client software instead of modifying memory structures or registry settings directly.

Windows Server 2003 allows administrators to take advantage of side-by-side loading with existing components as well as using a feature called DLL redirection. The operating system changes its default method of locating components if it finds a special file in the directory with the application that is loading the component. The file itself is empty but is specifically named to match the application executable name and has a .local suffix. For instance, **myapp.exe** would have an empty file next to it named **myapp.exe.local**. When Windows Server 2003 encounters this file, it looks for a requested component in the directory where the calling application is located or in the subdirectories below it. It will use the version of the component it finds there, no matter what path the system has registered for that component. If it cannot find a version in the directory structure of the application, the system will revert to using the path that is registered. Applications without a .local file continue to use the system registered path.

This method works for most components, but it needs to be tested to ensure that applications that use different versions can actually coexist. Components that store global state in registry keys that are not tied to a particular application or application version or shared memory structures may not operate correctly side by side. Sometimes this can be overcome just by not running simultaneously applications that load different versions, but each administrator must decide if that is acceptable to his or her user base. Other components may use relative paths to access system resources or other components, assuming that they are located in a particular directory. Some things can be safely moved or copied to satisfy this, but Windows system components, especially those protected by Windows file protection, should never be moved.

Managing Applications

Windows includes the Windows Management Instrumentation (WMI) component to manage or monitor applications. WMI is a component of the Windows operating system and is the Microsoft implementation of Web-based Enterprise Management (WBEM), which is an industry initiative to develop a standard technology for accessing management information in an enterprise environment. WMI uses the Common Information Model (CIM) industry standard to represent systems, applications, networks, devices, and other managed components. You can use WMI to automate administrative tasks in an enterprise environment.

Notes

- Additional information is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wmisdk/wmi/wmi_reference.asp.
- Additional information on WBEM implementations on UNIX is available at http://www.openwbem.org/.

Testing Activities

This section discusses the testing activities designed to identify and address potential solution issues before deployment. Testing starts when you begin developing the solution and ends when the testing team certifies that the solution components meet the schedule and quality goals established in the project plan.

Testing in migration projects involving infrastructure services is focused on finding discrepancies between the behavior of the original application, as seen by its clients, and the behavior of the newly migrated application. All discrepancies must be investigated and fixed.

In the Developing Phase, the testing team executes the test plans for acceptance tests on the application submitted for a formal round of testing on the test environment. The testing team assesses the solution, makes a report on its overall quality and feature completeness, and certifies that the solution features, functions, and components address the project goals.

The inputs required for the Developing Phase include:

- Functional specifications document.
- A feature-complete application, which has been unit tested.

The documents that are used during the Developing Phase include:

- **Test plan**. The test plan is prepared during the Planning Phase. It should describe in detail everything that the test team, the program management team, and the development team must know about the testing to be done.
- **Test specification**. The test specification conveys the entire scope of testing required for a set of functionality and defines individual test cases sufficiently for the testers. It also specifies the deliverables and the readiness criteria.
- **Test environment**. The test environment is an exact replica of the production environment; it is used to test the application under realistic environments. It also describes the software, hardware, and tools required for testing purposes.

- **Test data**. The test data is a set of data for testing the application. Test data is usually a diverse set of data that helps test the application under different conditions.
- **Test report**. The test report is an error report of the tests done. It includes a description of the errors that occurred, steps to reproduce the errors, severity of the errors, and names of the developers who are responsible for fixing them.

  The test report is updated during the Stabilizing Phase and is also one of the outputs of this phase, along with the tested and stabilized application.

The key deliverables of the Developing Phase include:

- Application ready to be deployed in the production environment.
- Application source code.
- Project documentation and user manual.
- Test plan, test specification, and test reports.
- Release notes.
- Other project-related documents.

Testing begins with a code review of the application and unit testing. In the Developing Phase, the application is subjected to various tests. The test plan organizes the testing process into the following elements:

- Code component testing
- Integration testing
- Database testing
- Security testing
- Management testing

You can test the migrated application in all the scenarios using a defined testing strategy. Although each test has a different purpose, together they verify that all system elements are properly integrated and perform their allocated functions.

Code Component Testing

A component may be a class or a group of related classes performing a similar task. Component testing is the next step after unit testing. Component testing is the process of verifying a software component with respect to its design and functional specifications.

Component testing in a migration project is the process of finding the discrepancies between the functionality and output of components in the Windows application and the original UNIX application. Basic smoke testing, boundary conditions, and error test cases are written based on the functional specification of the component.

The code component testing round tests the components for the following:

- Functionality
- Input and output, interactions within and with other components
- Stress testing
- Performance

The test cases for component testing cover, either directly or indirectly, constraints on their inputs and outputs (pre-conditions and post-conditions), the state of the object, interactions between methods, attributes of the object, and other components. The code component testing requires the following inputs:

- **Test plan and specification**. It provides the test cases.
- **System requirements**. These are used to determine the required behaviors for individual domain-level classes. The use case model is also used to determine which parts of a component must be tested for vulnerabilities.

- **Specifications of the component**. The specifications are used to build the functional test cases. Information on the component inputs, outputs, and interactions with other components can be derived from here.
- **Design document**. The actual implementation of the design provides the information necessary to construct the structural and interaction test cases.

Components must also be stress tested. Stress testing is the process of loading the component to the defined and undefined limits. Each component must be stressed under a load to ensure that it performs well within a reasonable performance limit.

System CPU and memory usage per component can also be measured and monitored to determine the performance of individual components. For this, you can use such tools as the Windows Performance Monitor. For more information, refer to the "Testing and Optimization Tools" section of Chapter 9, “Stabilizing Phase” of this volume.

Integration Testing

Integration testing involves testing the application as a whole, with all the components of the application put together. Component testing is done during the testing performed in the Developing Phase. Integration testing is the process of verifying the application with respect to the behavior of components in the integrated application, interaction with other components, and the functional specifications of the application as a whole. Integration testing in a migration project is the process of finding discrepancies in the interaction between components and the behavior of components in the Windows application and the original UNIX application.

Integration testing tests the components for:

- Functionality: behavior of the application as a whole and the individual components after integration.
- Input and output: interactions within and with other components.
- Response to various types of stresses.
- Performance.

Test cases for integration testing directly or indirectly include functionality of the components, constraints on their inputs and outputs (pre-conditions and post-conditions), the state of the object, interactions between components, attributes of the object, and other components. Inputs required for integration testing include:

- **Test plan**. It provides the details of testing the application.
- **Test specification**. It is used to determine the required behaviors for individual domain-level classes. The use case model is also used to determine which parts of the application must be tested for vulnerabilities.

The application must also be stress tested. Stress testing is the process of loading the application to the defined and undefined limits to ensure that it performs well within a reasonable performance limit.

System testing is also performed after completion of integration testing. System testing is the process of ensuring that the integrated application is compatible with all platforms and to test against its requirements. The system CPU and memory usage for the application can also be measured and monitored to determine their performance. For this, you can use such tools as the Windows Performance Monitor.

Note For more information, refer to the "Testing and Optimization Tools" section of Chapter 9, "Stabilizing Phase."

Database Testing

The database component is a critical piece of any data-enabled application. In a migration project, the database may be the same or may have been replaced by another database. In both cases, data must be migrated to the respective database on Windows. Testing of a migrated database includes testing of:

- Migrated procedural code.
- Data integration with heterogeneous data sources (if applicable).
- Customized data transformations and extraction.

Database testing also involves testing at the data access layer, which is the point at which your application communicates with the database. Database testing in a migration project involves:

- Testing the data and the structure and design of the migrated database objects.
- Testing the procedures and functions related to database access.
- Security testing, which tests the database to guarantee proper authentication and authorization so that only users with the appropriate authority access the database. The database administrator must establish different security settings for each user in the test environment.
- Testing of data access layer.
- Performance testing of data access layer.
- Manageability testing of the database.

An application maintains the following three databases, which are replicas of each other:

- **Development database**. This is where most of the testing is carried out.
- **Deployment database (or integration database)**. This is where the tests are run prior to deployment to ensure that the local database changes are applied.
- **Live database**. This has the live data; it cannot be used for testing.

Database testing is done on the development database during development, and the integrated application is tested using the deployment database.

Security Testing

Security is about controlling access to a variety of resources, such as application components, data, and hardware. Security testing is performed on the application to ensure that only users with the appropriate authority are able to use the applicable features of the application. Security testing also involves testing the application from the point of view of providing the same security features and measures that were provide by the original application.

To ensure that the application is secure, most security measures use the following four concepts:

- **Authentication**. This is the process of confirming the identity of the users, which is one layer of security control. Before an application can authorize access to a resource, it must confirm the identity of the requestor.
- **Authorization**. This is the process of verifying that an authenticated party has the permission to access a particular resource, which is the layer of security control following the authentication.
- **Data protection**. This is the process of providing data confidentiality, integrity, and nonrepudiability. Encrypting the data provides data confidentiality. Data integrity is achieved through the use of hash algorithms, digital signatures, and message authentication codes. Message authentication codes (MAC) are used by technologies such as SSL/TLS to verify that data has not been altered while in transit.

- **Auditing**. This is the process of logging and monitoring events that occur in a system and are of interest to security.

  Note For more information, refer to "Set event logging options" on the TechNet Web site at

  http://technet2.microsoft.com/WindowsServer/en/Library/0473658c-693d-4a06-b95b-ebe8a76648a91033.mspx.

The systems engineer establishes different security settings for each user in the test environment. Network security testing is performed to guarantee that the network is secure from unauthorized users. To minimize the risks associated with unchecked errors on the system, you should know the user context in which system processes run, keeping to a minimum the privileges that these accounts have, and log their access to these accounts. Active monitoring can be accomplished using the Windows Performance Monitor for real-time feedback.

All security settings and security features of the application must be documented properly.

Notes

More information about security testing is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxcontestingforsecurability.asp.

More information on how to make your code secure is available at

http://msdn.microsoft.com/security/securecode/.

More information on "Secure Coding Guidelines for the .NET Framework" is available at

http://msdn.microsoft.com/security/securecode/bestpractices/default.aspx?pull=/library/en-us/dnnetsec/html/seccodeguide.asp.

Management Testing

Testing for manageability involves testing the deployment, maintenance, and monitoring technologies that you have incorporated into your migrated application.

Following are some important testing recommendations to verify that you have developed a manageable application:

- **Test Windows Management Instrumentation (WMI)**. WMI can provide important information about your application and the resources it uses. During the design of your application, you made certain decisions about the types of WMI information that must be provided. These might include server and network configurations, event log error messages, CPU consumption, available disk space, network traffic, application settings, and many other application messages. You must test every source of information and be certain you can monitor each one.
- **Test Network Load Balancing (NLB) and cluster configuration**. You can use Application Center 2000 clustering to add a front-end or back-end server while the application is still running. After installing new server hardware on the network, use your monitoring console to replicate the application image and start the server. The new server should automatically begin sharing some of the workload. You can set up the Application Center 2000 Performance Monitor (PerfMon) to track multiple front-end Web servers. After setting up PerfMon, make some requests to generate traffic. PerfMon will show you that there is an increase in traffic in the back-end servers and that the workload is evenly spread across the front-end computers.

  Note Additional information about Application Center 2000 is available at

  http://www.microsoft.com/applicationcenter/.
- **Test change control procedures**. An important part of application management is the handling of both scheduled and emergency maintenance changes. Test and validate all of the change control procedures including the automated and manual processes. It is especially important to test all people-based procedures to ensure that the necessary communication, authority, and skills are available to support an error-free change control process.

  Note Additional information on testing for manageability is available at

  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxcontestingformanageability.asp.

Interim Milestone: Internal Release *n*

The project needs interim milestones that can help the team measure their progress in the actual building of the solution during the Developing Phase. Each internal release signifies a major step toward the completion of the solution feature sets and achievement of the associated quality level. Depending on the complexity of the solution, any number of internal releases may be required. Each internal release represents a fully functional addition to the solution’s core feature set, indicating that it is potentially ready to move on to the Stabilizing Phase.

Closing the Developing Phase

Closing the Developing Phase requires completing a milestone approval process. The team documents the results of different tasks that it has performed in this phase and obtains a sign-off on the completion of development from the stakeholders (including the customer).

Key Milestone: Scope Complete

The Developing Phase culminates in the Scope Complete Milestone. At this milestone, all features are complete and the solution is ready for external testing and stabilization. This milestone is the opportunity for customers and users, operations and support personnel, and key project stakeholders to evaluate the solution and identify any remaining issues that must be addressed before beginning the transition to stabilization and, ultimately, to release.

Key stakeholders, typically representatives of each team role and any important customer representatives who are not on the project team, signal their approval of the milestone by signing or initialing a document stating that the milestone is complete. The sign-off document becomes a project deliverable and is archived for future reference.

Now the team must shift its focus to verify that the quality of the solution meets the acceptance criteria for release readiness. The next phase, the Stabilizing Phase, describes the activities—for example, user acceptance testing (UAT), regression testing, and conducting the pilot—required to achieve these objectives.

Chapter 9: Stabilizing Phase

This chapter covers the strategy suggested for stabilizing an application that has been migrated from UNIX to the Microsoft® Windows® operating system. The Stabilizing Phase involves testing the application for the expected functionality and improving the quality of the application to meet the acceptance criteria set for the project.

This chapter describes the objectives of testing in the Stabilizing Phase. It introduces testing processes, as well as methodology and tools that you can employ to test applications with different architectures. It includes a set of job aids that can be used to develop test checklists to define the actions that you must take to ensure that the solution has been adequately tested and approved before its release. These job aids are specific to different architectures and also provide information on tuning the applications.

Goals for the Stabilizing Phase

The primary goal of the Stabilizing Phase is to improve the quality of the solution so that it meets the acceptance criteria and can be released to the production environment. During this phase, the team tests the feature-complete migrated application by subjecting it to various tests such as user acceptance testing (UAT), regression testing, and bug tracking based on the application's requirements. The resulting build must demonstrate that it meets the defined quality and performance level and be ready for full production deployment.

Testing during the Stabilizing Phase is an extension of the testing that is conducted during the development of the application in the Developing Phase. Testing in the Stabilizing Phase tests usage and operation of the application under realistic environment conditions. Test plans include a comparison of the migrated application's functionality with that provided by the original application. Test plans must also include test cases for testing the new features added to the application.

After the build is stabilized, the solution is deployed. The Stabilizing Phase ends with the Release Readiness Approved Milestone, indicating that the team and customers agree that all outstanding issues have been addressed.

Major Tasks and Deliverables

Table 9.1 describes the tasks that must be completed during the Stabilizing Phase and lists the roles responsible for achieving them.

Table 9.1. Major Stabilizing Phase Tasks and Owners

Major Tasks	Owners
Testing the solution The team executes the test plans that were created during the Planning Phase and enhanced and executed during the Developing Phase. Testing includes comparing the test results of the parent application with the migrated application, as well as testing the application from different perspectives.	Test
Resolving solution defects The team triages the identified defects and resolves them. New tests are developed to reproduce issues reported from	Development and Test

Major Tasks	Owners
other sources. The new test cases are integrated into the test suite.	
Conducting the solution pilot This task involves setting up the deployment environment and the migrated application on the staging area to test the application before it is deployed. The team moves a solution pilot from the development area to a staging area in order to test the solution with actual users and real scenarios. The solution pilot is conducted before starting the Deploying Phase.	Release Management
Closing the Stabilizing Phase The team documents the results of the tasks performed in this phase and solicits management approval at the Release Readiness Approved Milestone meeting.	Project

Table 9.2 lists the tasks described in Table 9.1, but focuses on the tasks from the perspective of the team roles. The primary team roles driving the Stabilizing Phase are Test and Release Management.

Table 9.2. Role Cluster Focuses and Responsibilities in the Stabilizing Phase

Role Cluster	Focus and Responsibility
Product Management	Execute communications plan. Launch test phase.
Program Management	Track project. Triage bugs.
Release Management	Prepare for the deployment of the application. Set up the production environment.
Development team	Triage bugs and resolve them. Optimize code. Reconfigure hardware or service.
User Experience	Stabilize user documentation and training materials.
Test	Define test goals and generate a master test plan (MTP). Generate build and triage plan. Generate detailed test plan (DTP). Review DTP and detailed test cases (DTC). Track test schedule. Review bugs entered in the bug-tracking tool and monitor their status during triage meeting. Generate weekly status reports. Escalate issues that are blocking progress, review impact analysis, and generate change management document. Ensure that appropriate level of testing is achieved for a particular release. Lead the actual build acceptance test (BAT) execution. Execute test cases and generate test report.

Testing the Solution

This section describes the testing activities that are performed in the Stabilizing Phase. In the Stabilizing Phase, because all features and functions of the solution are now complete and all solution elements have been built, testing is performed on the solution as a whole, not just on individual components. The testing that began during the Developing Phase according to the test plan created during the Planning Phase continues with further testing, tracking, documentation, and reporting activities during the Stabilizing Phase. This mainly involves user acceptance testing (UAT) and regression testing as explained in the next subsections in detail.

User Acceptance Testing

The emphasis on user acceptance testing (UAT) during the Stabilizing Phase is to ensure that the migrated solution meets the business needs. UAT is performed on a collection of business functions in a production environment after the completion of functional testing. This is the final stage in the testing process before the system is accepted for operational use. It involves testing the system with data supplied by the actual user or customer instead of the simulated data developed as part of the testing process. The UAT helps to validate the solution for the overall user requirements and also determines the release readiness status of the system. Running a pilot for a select set of users helps to identify areas where users have trouble understanding, learning, and using the solution.

For migration projects, UAT involves testing the migrated application and identifying its defects. These defects are addressed and regression test is conducted for each fixed defect to ensure that the fix doesn’t break any other functionality of the migrated application. The UAT Summary confirms that the solution meets the customer’s acceptance criteria, thereby furthering customer acceptance of the solution.

Regression Testing

Regression testing refers to retesting previously tested components and functionality of the system to ensure that they function properly even after a change has been made to parts of the system. For migration projects, this is the most important class of tests. As defects are discovered in a component, modifications should be made to correct them. This may require retesting of other components or the entire solution.

Regression testing helps in the following areas:

- To ensure that no new problems are introduced and that the operational performance has not been degraded because of modifications.
- To ensure that the effects of the changes are transparent to other areas of the application and other components that interact with the application.
- To modify the original test data and test cases from other testing activities.

Resolving Solution Defects

In order to resolve defects, you must reproduce and test them in the test environment. Each reproduced defect in the test environment should be tracked for its status and severity. An important aspect of such tests involves test tracking and reporting. Test tracking and reporting occurs at frequent intervals during the Developing and Stabilizing Phases. During the Stabilizing Phase, this reporting is driven by the bug count. Regular communication of the test status to the team and other key stakeholders ensures that the project runs smoothly. After fixing the defects, test cases and test data should be updated and integrated with the test suite.

Bug Convergence

Bug convergence is the point at which the team makes visible progress against the active bug count. At bug convergence, the rate of bugs resolved exceeds the rate of bugs found, thus the actual number of active bugs decreases. After bug convergence, the number of bugs should continue to decrease until the zero bug bounce task, as explained in the next sections.

Interim Milestone: Bug Convergence

Bug convergence tells the team that most of the bugs have been addressed and that the rate of bugs resolved is higher than the rate of new bugs found. This can be considered as the interim milestone and the migrated application can be considered for zero bug bounce verification.

Zero Bug Bounce

Zero bug bounce is the point in the project when development finally catches up to testing and there are no active bugs for the moment. After zero bug bounce, the number of bugs should continue to decrease until the product is sufficiently stable for the team to build the first release candidate.

Interim Milestone: Zero Bug Bounce

Achieving zero bug bounce is a clear sign that the solution is near to being considered a stable release candidate.

Release Candidates

After the first achievement of zero bug bounce, a series of release candidates are prepared for release to the pilot group. Each release is marked as an interim milestone.

Guidelines for declaring a build as a release candidate include the following:

- Each release candidate has all the required elements to qualify for release to production.
- The test period that follows determines whether a release candidate is ready to release to production or if the team must generate a new release candidate with appropriate fixes.
- Testing the release candidates, carried out internally by the team, requires highly focused, intensive efforts and concentrates heavily on discovering critical bugs.

Interim Milestone: Release Candidate

As each new release candidate is built, there should be fewer bugs reported, classified, and resolved. Each release candidate marks significant progress in the team’s approach toward deployment. With each new candidate, the team must focus on maintaining tight control over quality.

Interim Milestone: Preproduction Test Complete

Eventually, a release candidate is prepared that contains no defects. After this has occurred, no defects should be found within the isolated staging environment. At this stage, all testing that can be done before putting the migrated component into production has been completed.

Conducting the Solution Pilot

This section describes the best practices to adopt for conducting a pilot of the migrated application. This section provides you with information regarding various points to be considered when conducting a pilot and deciding the next steps after the pilot.

A pilot release is a deployment into a subset of the live production environment or user group. During the pilot, the team tests as much of the entire solution as possible in a true production environment. Depending on the context of the project, the pilot can take various forms:

- In an enterprise, a pilot can be a group of users or a set of servers in a data center.
- For migration projects, the pilot might involve testing the most demanding application or database that is being migrated with a sophisticated group of users who can provide helpful feedback.

The common element in all piloting scenarios is testing under live conditions. The pilot is not complete until the team ensures that the solution is viable in the production environment and that the solution is ready for deployment.

Some of the best practices that should be followed when conducting a pilot are:

- Before beginning a pilot, the team and the pilot participants must clearly identify and agree upon the success criteria for the pilot. These should map back to the success criteria for the development effort.
- Any issues identified during a pilot must be resolved either by further development, by documenting resolutions and workarounds for the installation team and production support staff, or by incorporating them as supplemental material in training or Help documentation.
- Before the pilot is started, a support structure and an issue-resolution process must be in place. This may require that the support staff receive training in the application area that is being piloted.
- In order to determine any issues and confirm that the deployment process will work, it is necessary to implement a trial run or a rehearsal of all the elements of the deployment prior to the actual deployment.

After you collect and evaluate the pilot data, a corresponding strategy should be selected based on the findings from the analysis of pilot data. The next strategy could be one of the following:

- **Stagger forward.** Deploy a new release to the pilot group.
- **Roll back.** Execute the rollback plan and revert the pilot group to the stable state they had before the pilot started.
- **Suspend.** Suspend the entire pilot.
- **Fix and continue.** If you find an issue during the pilot, fix the issue and continue with the next steps.
- **Proceed.** Advance to the Deploying Phase.

After the pilot has been completed, the pilot team must prepare a report detailing each lesson learned and how new information was incorporated and issues were resolved.

Interim Milestone: Pilot Complete

This milestone signifies that the pilot has been successfully completed and that the team is ready to proceed to the Deploying Phase.

Closing the Stabilizing Phase—Release Readiness Approved

The Stabilizing Phase culminates with the Release Readiness Approved Milestone. The team builds a release candidate with all major defects fixed as per the quality policy of the organization. All rounds of testing must be done before moving the migrated component into the production environment. When all test plans are executed and test cases are satisfied, the migrated application is ready to be moved to the production environment after the release is approved with a formal sign-off.

Key stakeholders, typically representatives of each team role and any important customer representatives who are not on the project team, signal their approval of the milestone by signing or initialing a document stating that the solution is complete and approved for release. The sign-off document becomes a project deliverable and is archived for future reference.

The performance of the application following deployment in the production environment is a key criterion in indicating a successful application migration. The following sections will help you to optimize the performance of the application and the tools following deployment.

Tuning

This section discusses tuning of the solution in detail, including how to performance-tune the migrated application, and scaling up and scaling out of the application. In addition, the section discusses multiprocessor considerations for applications and network utilizations. You can use this information to identify the parameters that affect application performances and steps to consider in the scalability of applications.

Performance Tuning

Performance management starts with the gathering of a data baseline that indicates what system performance should look like. After establishing a baseline, it is used to evaluate the performance of the application. Performance problems typically do not become apparent until the application is placed under an increased load.

Measuring the performance of an application when placed under ever increasing loads determines the scalability of that application. When the performance begins to fall below the stated minimum performance requirements, you have reached the limit of scalability of the application. For more information about scaling, refer to the "Scaling Up and Scaling Out" section later in this chapter.

Performance tuning can be done in the following ways:

- Tuning the computer hardware by adding more memory, updating CPUs, adding disk controllers, or upgrading network controllers. This is the most efficient way and helps performance-tune the application as well.
- Application rearchitecture to remove bottlenecks such as poor threading and looping and checking for other loops that use too much CPU time. This step also helps considerably in performance tuning.

- Operating system parameter tuning, which involves adjusting the amount of page store and tweaking network stack parameters.
- Tuning the configurations on a database server, application server, or Web server.

In UNIX, performance is monitored using a type of kernel-level instrumentation, along with rudimentary tools for monitoring the CPU, disk, and memory usage. Windows Server 2003 is designed such that it exposes a great deal of performance data. Tools like Windows Performance Monitor (PerfMon) can be used to export detailed information about the processor, memory, disk, and network usage. Performance Monitor support is integrated throughout Windows. Administrators can gather a variety of performance data from many computers simultaneously.

UNIX kernels tend to have many configurable parameters that can be fine-tuned for specific applications. By contrast, the Windows kernel is largely self-tuned. The virtual memory, thread scheduling, and I/O subsystems all dynamically adjust their resource usage and priority to maximize throughput. The difference between these two approaches is that the UNIX approach is to tweak kernel parameters for maximum advantage in the benchmark, even if those tweaks affect the real-world performance, while the Windows approach is to let the kernel tune itself for whatever load is placed on it.

Notes

More information on improving performance is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/fastmanagedcode.asp.

More information on writing high-performance managed applications is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/highperfmanagedapps.asp.

Scaling Up and Scaling Out

Scalability is a measure of how easy it is to modify the application infrastructure and architecture to meet variances in utilization. As with other application capabilities, the decisions you make during the design and early coding phases largely dictate the scalability of your application.

Application scalability requires a balanced partnership between two distinct domains: software and hardware. Because scalability is not a design concern of stand-alone applications, the applications discussed here are distributed applications.

Scaling up involves achieving scalability with the use of better, faster, and more expensive hardware to move the processing capacity limit from one part of the computer to another. Scaling up includes adding more memory, adding more or faster processors, or just migrating the application to a more powerful, single computer. Typically, this method allows for an increase in capacity without requiring changes to source code. However, adding CPUs does not add performance in a linear fashion. Instead, the performance gain curve slowly tapers off as each additional processor is added.

Scaling out distributes the processing load across more than one server by dedicating several computers to a common task. In this, the fault tolerance of the application is increased. Scaling out also presents a greater management challenge because of the increased number of computers.

Developers and administrators use a variety of load-balancing techniques to scale out with the Windows platform. Load balancing allows an application site to scale out across a cluster of servers, making it easy to add capacity by adding replicated servers. It provides redundancy, giving the site failover capabilities so that it remains available to users even if one or more servers fail or are taken down.

Scaling out provides a method of scalability that is not hampered by hardware limitations. Each additional server provides a near linear increase in scalability.

The key to successfully scaling out an application is location transparency. If any of the application code depends on knowing which server is running the code, location transparency has not been achieved and scaling out will be difficult. This situation requires code changes to scale out an application from one server to many, which is seldom an economical option. If you design the application with location transparency in mind, scaling out becomes an easier task.

Notes

More information on scaling is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconmanageabilityoverview.asp.

Microsoft Application Center 2000 reduces the complexity and the cost of scaling out. More information on "Application Center 2000" is available at

http://www.microsoft.com/applicationcenter/default.mspx.

More information on scaling network-aware applications is available at

http://msdn.microsoft.com/library/default.asp?url=/msdnmag/issues/1000/Winsock/toc.asp.

Multiprocessor Considerations

Application performance improves by having multiple processors perform the same task. You can distribute the processing load across several processors.

Computationally intensive tasks are characterized by intensive processor usage with relatively few I/O operations. The ongoing challenge with these applications is to improve the performance. You can do this with a faster computer, a more efficient algorithm, and by improving the implementation or using more processors. You can improve the performance with the help of tuning techniques as well.

Using more processors can mean taking advantage of an SMP computer or by using distributed computing with multiple networked computers. However, adding CPUs does not add performance in a linear fashion. Instead, the performance gain curve slowly tapers off as each additional processor is added. For computers with SMP configurations, each additional processor incurs system overhead. After you have upgraded each hardware component to its maximum capacity, you will eventually reach the real limit of the processing capacity of the computer. At that point, the next step is to move to another computer.

Multiprocessor optimization can be achieved by making use of threads.

Note More information on multiprocessor optimizations is available at

http://msdn.microsoft.com/msdnmag/issues/01/08/Concur/.

Network Utilizations

Network resources, such as available bandwidth and latency, must be predicted and managed on computers and devices throughout the network.

Optimal network utilization is achieved with cooperation among end nodes, switches, routers, and wide area network (WAN) links through which data must pass. Preferential treatment must be given for certain data as it traverses through the network in order to service certain components better during congestion. There are tools that help analyze network traffic, provide network statistics and packet information, and thereby better use the network by analyzing areas of congestion.

Quality of Service (QoS), an industry-wide initiative, achieves a more efficient use of network resources by differentiating between data subsets. Windows 2000 implements QoS by including a number of components that can cooperate with one another.

Note More information on QOS on Windows is available at

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/qos/qos/qos_start_page.asp.

Note Network Monitor captures network traffic for display and analysis. More information on Network Monitor is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/netmon/netmon/network_monitor.asp.

Note Network Probe is another tool for traffic-level network monitoring and for analysis and visualization. More information on Network Probe is available at http://www.objectplanet.com/probe/.

Testing and Optimization Tools

This section lists some of the useful tools that can be used for testing and monitoring your applications.

Visual Studio .NET 2003 Tools

Microsoft Visual Studio® .NET 2003 includes tools for analyzing the performance of applications. These include:

- **Process Viewer (Pview)**. The PView process viewer uses dialog boxes to view and modify running processes and their threads. PView can monitor:
  - Memory usage of process, threads, and individual DLLs.
  - CPU time used by processes and threads.
  - How an application or the system runs with different system priorities.

  PView features provide powerful tools with which you can monitor processes of an application and threads at different priorities. More information about the Process Viewer is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcsample98/html/vcsmppviewer.asp.
- **Spy++.** Spy++ shows a graphical view of the processes of the system, threads, windows, and window messages. More information about Spy++ is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcug98/html/_asug_overview.3a_.spy.2b2b.asp.
- **DDESpy.** DDESpy monitors dynamic data exchange (DDE) activity in the operating system. More information about DDESpy is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcsample98/html/vcsmppviewer.asp.

Platform SDK Tools

Platform SDK tools includes debugging tools, file management tools, performance tools, and testing tools. These tools are available with the latest Platform SDK.

Debugging Tools

Platform SDK includes the following debugging tools:

- **Debug Monitor (DBMon)**. The Debug Monitor runs in its own console window and displays messages sent by your application. More information about the Debug Monitor is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/debug_monitor.asp.
- **Symbolic Debugger (NTSD)**. NTSD is a symbolic debugger that enables you to debug user-mode applications. You can display and execute program code, set breakpoints, and examine and change values in memory. More information about the Symbolic Debugger is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/symbolic_debuggers.asp.
- **Windows Debugger (WinDbg)**. The WinDbg debugger is a powerful graphical tool that allows you to debug applications on Microsoft Windows. You can use the integrated text editor to edit your source code. WinDbg can also be used to debug service applications and kernel-mode drivers. More information about the Windows Debugger is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/windbg_debugger.asp.

File Management Tools

- **WinDiff**. WinDiff is used to compare files and display the results graphically. More information about WinDiff is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/windiff.asp.

Performance Tools

Performance tools can be used to measure application performance and resolve some performance issues. Platform SDK includes the following performance tools:

- **Bind**. Bind minimizes application load time by binding your executable with all of your DLLs, plus the system DLLs. More information about Bind is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/bind.asp.
- **Extensible Performance Counter List (ExCtrLst)**. The extensible counter list tool is used to obtain information about the extensible performance counter dynamic-link libraries on a computer. More information about ExCtrLst is available at http://www.microsoft.com/downloads/details.aspx?FamilyID=7ff99683-b7ec-4da6-92ab-793193604ba4&DisplayLang=en .
- **Performance Meter (PerfMtr)**. PerfMtr can display a variety of system performance information. More information about the PerfMtr is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/perfmtr.asp.
- **Performance Monitor (PerfMon)**. Windows Performance Monitor can simultaneously collect performance data from any number of network computers, then display it as a graph, format it as a tabular report, or log it for later analysis. Performance Monitor support is integrated throughout Windows. More information about PerfMon is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/perfmtr.asp.
- **PStat**. PStat lists statistics for each process. More information about PStat is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/pstat.asp.
- **Virtual Address Dump (VADump)**. Virtual Address Dump creates a listing that contains information about the memory usage of a specified process. More information about VADump is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/vadump.asp.

Testing Tools

- **Process Fault Monitor (PfMon)**. The Process Fault Monitor displays the faults that occur while executing a process. PFMon can start the application for you or attach to a running process. More information about PfMon is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/pfmon.asp.

Other Commonly Used Tools

This section lists other commonly used tools that are useful in testing and monitoring applications.

Monitoring Tools

- **Diskmon**. This tool captures all hard disk activity or acts such as a software disk activity light in your system tray. This tool is available for download at http://www.sysinternals.com/ntw2k/freeware/diskmon.shtml.
- **Filemon**. This monitoring tool allows you to view all file system activity in real-time. This tool works on all versions of Windows NT, Windows 2000, Windows Server 2003, and Windows XP. It also works with the Windows XP 64-bit edition. This tool is available for download at http://www.sysinternals.com/ntw2k/source/filemon.shtml.
- **PMon**. This is a Windows NT GUI/device driver program that monitors process and thread creation and deletion, as well as context swaps if it is running on a multiprocessing or checked kernel. This tool is available for download at http://www.sysinternals.com/ntw2k/freeware/pmon.shtml.

- **Portmon**. You can monitor serial and parallel port activity with this advanced monitoring tool. It knows about all standard serial and parallel IOCTLs and even shows you a portion of the data being sent and received. This tool is available for download at http://www.sysinternals.com/ntw2k/freeware/portmon.shtml.
- **Regmon**. This monitoring tool allows you to view all registry activity in real-time. This tool is available for download at http://www.sysinternals.com/ntw2k/source/regmon.shtml.
- **TCPView**. You can view all the open TCP and UDP endpoints. TCPView even displays the name of the process that owns each endpoint. This tool is available for download at http://www.sysinternals.com/ntw2k/source/tcpview.shtml.
- **Task Manager**. Task Manager provides run-time information on processes. The Task Manager tool is available as part of Windows.

Testing Tools

- **WinRunner**. WinRunner helps in GUI capture and playback testing for Windows applications. More information on WinRunner is available at http://www.mercury.com/us/products/quality-center/functional-testing/winrunner/.
- **Silktest**. Silktest is an object-oriented software testing tool for Windows applications. More information on Silktest is available at http://www.segue.com/products/functional-regressional-testing/silktest.asp.
- **LoadRunner**. LoadRunner is an automated client/server system testing tool that provides performance testing, load testing, and system tuning for multiuser applications. More information on LoadRunner is available at http://www.mercury.com/us/products/performance-center/loadrunner/.
- **Rational Robot Automated Test**. Rational Robot Automated Test provides automated functional, regression, and smoke tests for e-applications. More information on Rational Robot is available at http://www-306.ibm.com/software/rational/.
- **Microsoft Application Center Test**. Designed to stress test Web servers and analyze performance and scalability problems with Web applications, including Active Server Pages (ASP) and the components they use. It simulates a large group of users by opening multiple connections to the server and rapidly sending HTTP requests. More information on Microsoft Application Center Test is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/act/htm/actml_main.asp.

Source Test Tools

- **Purify**. Purify is a run-time error and memory leak detector. More information on Purify is available at http://www-306.ibm.com/software/sw-bycategory.

Tools for win64:

- **VTune Performance Analyzer**. Intel VTune Analyzers help locate and remove software performance bottlenecks by collecting, analyzing, and displaying performance data from the system-wide level down to the source level. More information on VTune Performance Analyzer is available at http://www.intel.com/software/products/vtune/.

Further Reading

For more information, refer to:

- “Testing Software Patterns” on the MSDN Web site at http://msdn.microsoft.com/practices/guidetype/Guides/default.aspx?pull=/library/en-us/dnpag/html/tsp.asp.
- Tools are also available in .NET for creating components, which can be used to monitor system resources.

  For information about using event logs, performance counters, and services, refer to "System Monitoring Components" on the MSDN Web site at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vborisystemmonitoringwalkthroughs.asp.

Index